IDE Extension for Reasoning About Energy Consumption

Jacob Ruberg Nørhave,
Casper Susgaard Nielsen,
Anne Benedicte Abildgaard Ejsing

June 2021

Summary

In recent years the energy consumption of software and information and communications technologies (ICT) systems have become an ever increasing environmental and societal concern. It has been reported that the total energy consumption of data centres in 2018 amounted to approximately 200 TWh which corresponds to 1% of the world's total energy consumption [1]. Furthermore, this represents a 6% increase compared with 2010 [1]. However, software developers often neglect the issue of considering the energy consumption impact of their software [2].

We thus decide to create an IDE extension for Visual Studio Code to aid developers in reasoning about the energy consumption of their programs. We provide different types of energy consumption estimates to the user, being through static analysis and by dynamically executing the code and measuring the energy using RAPL. The static techniques consists of using machine learning and creating an energy model.

We create a prototype of the extension and implement the extension accordingly. The extension is implemented as a Visual Studio Code WebView that conforms to the general design of Visual Studio Code. From the user interface we allow the user to choose for which programs or methods they want energy consumption estimates. They can also chose how they want to obtain the measurements, be it statically using the energy model or machine learning or dynamically using RAPL. For the extension to get the energy estimates we create a microservice architecture where we include microservices for each estimation approach. We also create a microservice for our interpreter. In this way, the extension can communicate with the microservices using HTTP calls as

needed, and the microservices for the estimation approaches can communication with the interpreter microservice as needed.

For the energy model and the machine learning we explore different software abstractions to use as a basis for the estimations, being C# source code, CIL code, and machine code. We decide to use CIL code as this provides a restricted set of instructions while being machine independent. Then, to count CIL instructions, we create an interpreter for CIL code to count all CIL instructions in a program along with the number of times each CIL instruction are performed. For the machine learning models we investigate five different regression techniques to predict the energy consumption of programs. We decide to employ an experimental approach to determine which of these techniques performs best on the problem of this project. The machine learning models are trained on a set of 147 programs. For the per-instruction energy modelling approach, we used runtime code generation in C#, using the `System.Reflection.Emit` library, to emit individual instructions and thus measure their energy consumption. As instructions must be emitted to a method, we use the approach of creating `DynamicMethods`, which are also part of the `System.Reflection.Emit` library. We create `DynamicMethods` for each CIL instruction. Then the energy consumption of the methods were measured, and any dependencies subtracted. The measurements were then used for the implementation of the energy model estimation. Lastly, the dynamic estimation approach was created based on the measurement framework made in our previous work [3].

Based on the implemented estimation approaches, we determine the error of each static approach, where the dynamic estimates are used as the ground truth. This is because this approach measures the energy consumption of each benchmark program and provides statistically significant results. We calculate the error of a benchmark as the percentage from the ground truth. Based on these results, the non-linear machine learning estimates have a lower percentage deviation from the ground truth than the energy model and all of the linear machine learning models. Furthermore, the energy model has a lower percentage deviation from the ground truth than the linear models, except for lasso regression. The estimation approach with the least error is the random forest machine learning model, with a minimum of -7.49% and a maximum of 9.19%. The median of random forest is 1.06%, which indicates a slight over estimation of the energy consumption.

10th Semester
The Technical Faculty of IT and Design
Department of Computer Science
Selma Lagerlöfsvej 300
9220 Aalborg Øst
https://cs.aau.dk

Title: IDE Extension for Reasoning About Energy Consumption

Subject: Programming Technology

Project period:
P10, Spring 2021
01/02/2021 - 14/06/2021

Group No:
PT104F21

Group Members:
Jacob Ruberg Nørhave
Casper Susgaard Nielsen
Anne Benedicte Abildgaard Ejsing

Supervisor:
Bent Thomsen
Lone Leth Thomsen

Pages: 87

Abstract:

This project explains the process of creating an IDE extension for Visual Studio Code. This extension is used to reason about the energy consumption at both the program and function level. We implement three approaches for obtaining energy consumption estimates, being statically using machine learning and an energy model and dynamically using RAPL. For the static analysis we create an interpreter for CIL code to count the number of times each CIL instruction encountered during interpretation. To evaluate the accuracy of the static approaches we use the dynamically measured values as the ground truth and compares the statically obtained values to this. Based on these results, the non-linear machine learning creates better estimates than the energy model and the linear machine learning models. The energy model creates better estimates than the linear models, except for lasso regression. The estimation approach with the least error is the random forest machine learning model.

Preface

This project is a masters thesis on the Software Engineering education at Aalborg University. The project spans from the 1st of February 2020 to the 14th of June 2021.

This group and project is supervised by Bent Thomsen[1] and Lone Leth Thomsen[2], who has guided the project towards the learning goals of the semester.

In this report we create an IDE extension for Visual Studio Code that helps developers in reasoning about the energy consumption of their code. We create three approaches for estimating the energy consumption, being statically using machine learning or an energy model and dynamically using RAPL. We compare the accuracy of each approach to determine which performs the best. The raw data of the results is seen in the documents and files attached to this project.

Aalborg University, February 1st, 2021

[1]https://vbn.aau.dk/da/persons/110568
[2]https://vbn.aau.dk/da/persons/105801

List of Figures

2.2.1 Overview of the Common Language Infrastructure 9
2.4.1 Architecture of Visual Studio code [29] 25
2.4.2 An initial design for the User Interface for the extension 26
2.4.3 An initial design for the User Interface for the extension while estimating 26

3.1.1 Simple overview of the architecture of the IDE extension 28
3.2.1 A simple overview of the overall structure for counting instructions 31
3.2.2 An overview of the steps for the counting by interpretation approach 33
3.2.3 Distribution of scraped programs. 37
3.4.1 A view of the extension 51

4.4.1 Boxplots for Random-Forest and Support-Vector regression . . . 66
4.4.2 Boxplot of Lasso regression and the Energy model 68

List of Tables

3.1.1 API endpoints for the different microservices 29
3.2.1 Number of instructions for different benchmarks 38

4.2.1 Excerpt of the final energy results. All values are presented in µj and rounded to two decimals. For brevity the column names are shortned, such that `GT` means `Ground Truth`, `RF` means `Random Forest`, and `EM` means `Energy Model` 61
4.2.2 Excerpt of the final time consumption results. All values are presented in ms and rounded to two decimals. For brevity the column names are shortened, such that `RF` means `Random Forest`, and `EM` means `Energy Model` 64
4.3.1 Hardware specifications of test setup 64
4.4.1 Boxplot quartiles for the five machine learning approaches 67
4.4.2 Excerpt of the final time consumption results. All values are presented in ms and rounded to two decimals. For brevity the column names are shortened, such that `RF` means `Random Forest`, and `EM` means `Energy Model` 69

List of Listings

1 An example of a single line return using LINQ 9
2 A simple loop represented in CIL code 12
3 Empty Java application to measure the constant overhead 19
4 Java application that loads an integer 19
5 Creating a method based on an assembly, module, and type . . . 20
6 Creating a method using the dynamic approach 21
7 An example of a measure framework based on a standard unit testing framework 24
8 A snippet showing which types of data the `ActivateClass` contains 29
9 The simple representation of an instruction 32
10 A simplified view of the data for a method 32
11 The executor method for the interpreter 35
12 Snippet of script to train a regression model 38
13 Snippet of machine learning microservice 39
14 Creating and invocation of `DynamicMethod` 41
15 An example of measuring the energy consumption for the constant overhead, and the energy consumption of the CIL instruction `Ldc_I4` 42
16 Creating a random value based on a type 43
17 A snippet of the energy model 45
18 Snippet of the energy model microservice 46
19 Implementation of the MeasureAttribute 48
20 Shows how to get all methods of a class 50
21 `WebView` posting a message 52
22 `WebView Provider` posting a message 52
23 Switch-case of `WebView Provider` 53
24 `WebView Provider` communicating with microservices 54

25 Example of benchmark without the `Benchmark` library implemented. 59
26 Example of benchmark with the `Benchmark` library implemented. 59
27 Snippet of the script that computes the energy consumption using the static estimation techniques. 62
28 Snippet of the script that measures the time consumption while computing the energy consumption using the static estimation techniques. 63

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Related Work 2

2 Design **5**

2.1 High Level Design 5

2.2 Static Estimation 8

2.3 Dynamic Estimation 22

2.4 Extension 24

3 Implementation **27**

3.1 Architecture 27

3.2 Static Estimation 30

3.3 Dynamic Estimation 46

3.4 Extension 51

4 Experiments and Results **55**

4.1 Hypothesis 55

4.2 Measurement Approach 57

4.3 Test Setup 64

4.4 Results 64

5 Reflections **70**

6 Conclusion and Future Work **77**

6.1 Conclusion 77

6.2 Future work 80

Appendix

A Missing CIL instructions for the energy model

Chapter 1

Introduction

In recent years the energy consumption of software and information and communications technologies (ICT) systems have become an ever increasing environmental and societal concern. Thus, measures for reducing the energy consumption of software systems have been the source of much research in computer science. It has been reported that the total energy consumption of data centres in 2018 amounted to approximately 200 TWh which corresponds to 1% of the world's total energy consumption [1]. Furthermore, this represents a 6% increase compared with 2010 [1]. In addition to servers being responsible for a large portion of the world's total energy consumption, the energy consumption of personal computing and computer networks is also significant [4]. Besides, the energy consumption of software systems being an environmental concern, it is also a societal problem. This is because, companies want to reduce electricity expenses, and consumers want their battery-powered devices to last longer. This is especially relevant since it has been projected that there will be a large increase in the number of IOT devices in the next couple of years [5].

It is estimated that there are approximately 19 million software developers in the world in 2019 [6]. However, software developers often neglect the issue of considering the energy consumption impact of their software [2]. This neglect can be contributed to a lack of knowledge and tools for reasoning about energy consumption. Furthermore, performance requirements have often substituted energy consumption as the general consensus is that a faster programs uses less energy. This holds true in some cases, but in other cases a faster program can end up using more energy that a slower counterpart. Based on this, it is essential to develop methods with which developers can reason about the energy consumption of their software with ease.

1.1 Problem Statement

This sections presents the problem statement of this project and the motivation for solving said problem. The problem statement of this project is as follows:

How can an IDE extension be made to aid developers in reasoning about the energy consumption of their software

The problem statement is motivated by the lack of knowledge and tools surrounding the energy consumption of software. This is seen in both [2] and [7], where a set of software practitioners share their perspective on green software. From this, it is clear that we need tools to reason about the energy consumption of the software that are being developed.

The purpose of this report is to provide an answer to the problem given in the problem statement. To do this we develop an IDE extension for Visual Studio Code that allows developers to estimate the energy consumption of entire programs or individual methods in their code. First, we present work related to our problem in Section 1.2 where we elaborate on research related to energy measurement tools and energy estimation techniques. In Chapter 2 on page 5 we first create the high-level design of our IDE extension and the approaches we include for measuring the energy consumption. This is followed by a more in-depth view of the extension as well as the energy measurement approaches. After the design, we present the implementation based on the design in Chapter 3 on page 27. Followed by this, we create experiments to determine the accuracy of the implemented approaches, and we present the results in Chapter 4 on page 55. Lastly, we discuss the results in Chapter 5 on page 70. and conclude on the report in Chapter 6 on page 77.

1.2 Related Work

In this section we discuss research related to the work of this thesis. We examine what other tools exist to aid software developers in recognising the energy implications of their code, and thus encouraging the development of more energy efficient code. We also examine different strategies to measure and estimate the energy consumption of software programs.

1.2.1 Tools

Several related research papers revolve around the creation of tools to aid developers in reasoning about the energy consumption of their software. In [8] the authors create a function-level profiling tool that measure the energy consumption of code. This tool performs its program analysis at runtime and identify functions based on the trace of energy usage and the timestamps of programs execution events. Furthermore, in [9] the authors create a plugin for the Eclipse IDE that estimates the energy consumption at the code level, being both program, function, and line level. It does so by combining program analysis and per-instruction energy modelling. Another tool is developed in [10] that is capable of detecting and refactoring energy inefficient code in Android applications.

1.2.2 Estimation Techniques

Research papers have also investigated various techniques for estimating the energy consumption of software. First, [11] describes RAPL (Running average power limit), which is an on-chip power meter tool included on processors from Intel. RAPL provides the exact energy consumption of the power domains *Powerplane 0* (PP0), *Powerplane 1* (PP1), *DRAM*, and *Package* (PKG) with a temporal resolution of 1 ms. Other techniques involve estimating the energy consumption using machine learning, which is the approach seen in [12], where the prediction is based on software performance features such as cache hits and misses, context switches, and clock cycles. The machine learning model is thus independent of the programming language and it has an error rate of 6.8 %. Furthermore, several papers, including [13–15] estimates the energy consumption by creating an energy model based on the instructions. In [13] the authors create a probabilistic energy distributions of instructions and propose a model for composing instruction sequences using distributions, enabling worst case energy-consumption analysis on program basic blocks. They account for data-dependent instructions by providing random values to the instructions when measuring the energy consumption. In [14] the total energy consumption of a program is computed as $\sum_i(B_i\dot{N}_i) + \sum_{i,j}(O_{i,j}\dot{N}_{i,j}) + \sum_k E_k$, where B_i is the base cost of an instruction, weighted by the number of times, N_i, it is executed. $O_{i,j}$ is the overhead of executing instruction i followed by instruction j, weighted by the number of times, $N_{i,j}$, that sequence is executed. Lastly, E_k denote other inter-instruction effects such as stalls and cache misses. The last paper on energy modelling is [15], where an energy consumption model is created for an embedded Java virtual machine (JVM). They determine the constant overhead of the JVM and compute the energy cost of each Java Opcode by comparing the instructions and energy of small programs.

1.2.3 Preliminary Research about Energy Consumption

This report builds on research we have conducted during the making of the report *The Influence of Programming Paradigms on Energy Consumption* [3]. In this project we sought to provide an answer to the problem statement

How does the execution time and energy consumption compare, for programs written in different paradigms using the same multi-paradigm language?

To do this, we compiled an elaborate benchmark suite of problems written in the style of the procedural, object-oriented, and functional programming paradigm with all benchmarks implemented in C# as well as F#. We then created a benchmarking library for all the benchmarks to implement and to measure the consumed energy and running time of each benchmark, we implemented a library that uses the RAPL interface. The results of the project showed that the benchmarks written in the style of the procedural paradigm generally consumes the least energy while the functional paradigm performed worst overall. Throughout the current work we reference the above mentioned report when reusing elements already created such as the libraries or the general approach to measuring energy.

Chapter 2

Design

This chapter contains the design of the IDE extension we create. This includes a high-level design, provided in Section 2.1, where we give an overview of the initial considerations concerning the IDE extension. Then, in Section 2.2 on page 8 and Section 2.3 on page 22 we detail the two approaches, being static and dynamic estimation respectively, that we implement in the IDE extension. Lastly, in Section 2.4 on page 24 we design the functionality and user interface of the IDE extension.

2.1 High Level Design

This section serves to provide a high-level design and overview of the IDE extension that we create in this project. As stated in the problem statement (Section 1.1 on page 2), the goal of this project is to create an IDE extension that aids developers in recognising the energy implications of their software. To reason about the energy consumption of software we create and distinguish between two terms that collectively describes the what and how to perform measurements. The two terms are:

- Measurement Approach
- Energy Estimation Approach

The *measurement approach* describes the granularity of *what to measure*. This determines the amount of code to be measured, be it the entire program, individual functions, or lines of code. The *energy estimation approach* is the method used to obtain the energy consumption at the level of the measurement approach. This is the *how to measure*. In the following subsections we elaborate on the two considerations.

2.1.1 Measurement Approach

The first of the two categories is the *measurement approach*. This section introduces the granularity with which measurements can be performed. These include measuring at program, function, and line level.

Program level measurement, means that we estimate the energy consumption of the entire program, thus the granularity is coarse. It is not possible to determine *where* the energy is being used, however one can determine if one program is more energy efficient than another. With function level measurements we measure the energy consumption of individual functions within a program. This enables the developers to use the results to discover where the energy is being used, and compare functions relative to each other. However, several issues arise such as how to handle functions that are dependent on some input or functions that are either recursive or call other functions. Lastly, if measuring on line level, estimates are provided per line. This is the approach with the finest granularity and allows developers to reason about individual lines of code.

2.1.2 Energy Estimation Approach

The energy estimation approach denotes how the energy consumption of the software is estimated or measured. Generally, this can be done in one of two ways, being *statically* or *dynamically*.

If the energy consumption is to be estimated statically, then the related work have shown two approaches being either program analysis using an energy model or by creating a machine learning model based on properties of the code. To perform program analysis using an energy model each low-level or intermediate level instruction is mapped to their corresponding energy consumption. Then the total energy consumption is estimated based on the energy of each measured instruction. If, instead, a machine learning approach is chosen, features of the code that can be used to predict the energy consumption need to be identified. In [12] they use software performance features such as cache hits and misses and branch prediction success. However, other features can be used to predict the energy consumption as well. This can for example be based on the actual code at some level of abstraction. Both of these static approaches depend on knowledge of the energy consumption of each instruction, however as the energy per instruction is dependent on the processor on which it is measured ([16]), the static approaches are biased towards the processor on which the energy is measured.

If the energy consumption is measured dynamically a power meter can be used. Power meters can either be on-chip, such as Intel's RAPL, as described in [17], or external such as the Watts Up Pro, which is used in [10], [18], and [19]. On-chip power meters generally have higher spatial and temporal resolution than external power meters. This means, on-chip power meters can perform measurements on individual hardware components, such as the entire processor or the DRAM. However, external power meters work independently of the hardware and operating system. This is in contrast to Intel's RAPL, which is only available on Intel processors and the power measurements can only be accessed using the Linux operating system [17]. When measuring the energy consumption dynamically the code has to be run several times to obtain statistically significant results. Our preliminary research found in [3] describes both power meters and an approach for obtaining statistically significant results in more detail.

The two approaches, being static and dynamic estimation, complement each other in static being fast, but imprecise and dynamic being slow but precise. It is faster to obtain static estimates as the code does not need to be executed, whereas to obtain dynamic measurements the code must be executed a specified number of times for the measurements to be statistically significant. However, to obtain static estimates some precomputation is required, for example training a machine learning model or computing an energy model.

2.1.3 Summary

Based on the high-level discussion of the approaches from the previous sections, we include a static as well as a dynamic approach to energy estimation in the IDE extension. These two approaches are useful in different situations. The static approach being fast, but imprecise means that developers can get a quick overview of the energy consumption of the code. While the dynamic approach is slow, it presents more accurate readings of the energy consumption. The static analysis is achieved through the creation of an energy model as well as using machine learning. Furthermore, for the measurement approach we include program and function level measurements; thus excluding line level. We deem, that developers benefit the most from program and function level measurements as this allows them to reason about larger pieces of code relative to each other. Furthermore, when estimating the energy consumption, it is difficult to determine what constitutes a *line*. For example, considering a loop, should the energy be measured only for the lines of the loop body or should the line containing the loop condition also be measured? Also, in the C# language an entire program could be placed in *one* line, thus we deem the term *line* too vague for our project.

2.2 Static Estimation

As described in Section 2.1 on page 5 we consider two approaches to energy estimation through static analysis, one based on machine learning and another based on an energy model. However for both approaches we must first determine at what level of software abstraction the static analysis is performed. The following sections first discuss the different levels of software abstractions for static analysis and then elaborate on the two approaches for static estimation.

2.2.1 Levels of Software Abstractions for Static Analysis

To perform static analysis of an application, it is important to determine what level of software abstraction to perform the analysis on. For C#, there are several levels of abstractions which are viable for analysis. When executing a C# program, it is first compiled into an intermediate representation called the *Common Intermediate Language* (CIL), which is part of the *Common Language Infrastructure* (CLI). The CIL code is then managed by the *Common Language Runtime* (CLR). The CLR contains a Just-in-time (JIT) compiler to convert the CIL code into machine code. The machine code is then executed on the machine. An overview of this approach can be seen on Figure 2.2.1 on the following page. The approach leads to three scenarios where static analysis can be performed to estimate the energy consumption of a given program.

The first approach is to do the analysis of the C# code. An advantage of performing the analysis at this stage, is that no further actions have to be done

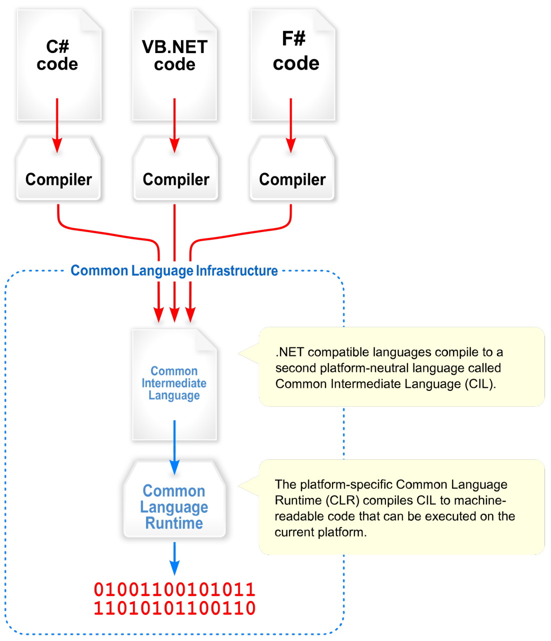

C# code
VB.NET code
F# code
Compiler
Compiler
Compiler
Common Language Infrastructure
Common Intermediate Language
.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).
Common Language Runtime
The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.
01001100101011
11010101100110

Figure 2.2.1: Overview of the Common Language Infrastructure - Deviousasti at Wikipedia

to the code to start the analysis. A disadvantage at this stage, is that C# is a high-level language with a plethora of ways to perform the same actions. Furthermore, it is possible to condense a lot of information into every single line of code. This is especially problematic with programming patterns such as fluid interfaces utilised by among other, the LINQ library. This fact is illustrated in the code in Listing 1, where one line of C# code performs all of the necessary actions to compute the sum of all even input values squared.

```
using System.Linq;
using System.Collections.Generic;

public int getSumOfEvenSquares(List<int> ints)
{
    return ints.Where(x => x % 2 == 0).Select(y => y * y).Sum();
}
```

Listing 1: An example of a single line return using LINQ

The second approach is to analyse the CIL code produced by compiling the C# code. The advantages of analysing at this step is that CIL is machine independent, like the C# code, while also being language independent. This means that other languages can be compiled to CIL and the analysis of the code would still work. Notable languages for the CLI are: C#, F#, and Visual Basic. The CIL stage also provides a reduced set of instructions compared to C# code, namely only 226. One disadvantage of performing the analysis at this stage, is that the code might not be executed exactly as written, because the JIT compiler optimises code in many ways [20].

The last approach is to do the analysis of the resulting machine code. The advantage of this approach is that the code is language independent, though there are several disadvantages to this approach. For one, the code has to be executed to know the exact machine code, which is produced by the JIT compiler. Second, is that the JIT compiler can generate different machine code for the same CIL code based on optimisations [20]. Third, is the fact that machine code is machine dependent. There are two factors to consider when working with machine code. The first is that the operating system can have specific APIs which are called during the program execution. If these APIs are not present on another operating system, then the execution will fail. The second factor is the influence of the CPU. Different CPU can run different instruction set architectures (ISA), which determine the set of instructions available when the code is executed on the hardware. It can therefore be much more difficult to analyse at this level.

We thus decide to perform the static analysis at the CIL instruction stage, as it provides the benefits of being both machine independent and language independent, while having a reduced instruction set to analyse.

2.2.2 Common Intermediate Language

As mentioned above CIL is a machine independent intermediate language. Specifically, it is a stack-based intermediate representation for the languages supported by the .NET platform. It is composed of a smaller set of instructions compared to the source languages, which also makes it more suitable for analysis. There are two categories of instructions in the CIL instruction set: *Base instructions* and *Object model instructions*. The former contains all of the instructions necessary for basic flow control and the basic type system, while the latter is concerned with object manipulation, which also includes custom user types.

With regard to determining the number of CIL instructions for a given CIL document, the most powerful category of instructions is the branch instructions, since they can alter the control flow either forwards or backwards in the code.

Branching in CIL exists in three different forms: *Unconditional*, *Conditional*, and *Compound-Conditional*. Unconditional branch instructions contain all of the instructions which always alters the control flow by selecting the next instruction to be executed. These instructions include:

1. br (Branch)
2. br.s (Branch short-form)
3. ret (Return)
4. call (Calls method)
5. callvirt (Call method associated with an object)

Conditional branch instructions include all of the branch instructions which have two modes of operation based on a certain condition. The first mode is to continue the sequential flow of execution. This happens if the condition is false. The second mode is to alter the control flow by branching to another instruction either backwards or forwards in the code. A subset of these instructions is seen below:

1. beq (Branch if equal)
2. blt (Branch if less than)
3. brfalse (Branch if value is 0)
4. brzero (Alias for brfalse)
5. brtrue (Branch if value is non-zero)

Lastly, compound-conditional branch instructions only contain a single instruction. The *switch* instruction is used to create a set of targets for branching based on some condition, called a *jump table*. The jump table can be used by indexing, using the result from the condition.

Many high-level constructs are implemented with a combination of the presented branch instructions. As an example, loops are generally created from a single

```
IL_0000: ldc.i4.0
IL_0001: stloc.0
IL_0002: ldc.i4.s 100
IL_0003: stloc.1
IL_0004: br.s IL_0008
IL_0005: ldloc.0
IL_0006: ldc.i4.1
IL_0007: add
IL_0008: stloc.0
IL_0009: ldloc.0
IL_000a: ldloc.1
IL_000b: clt
IL_000c: brtrue.s IL_0005
IL_000d: ldloc.0
IL_000e: ret
```

Listing 2: A simple loop represented in CIL code

unconditional branch. This branch changes the flow directly to the condition controlling a conditional branch instruction, jumping back to the start of the loop if the condition is true. Such a loop can be seen in Listing 2, where `IL_0004` is an unconditional branch starting the loop and `IL_000c` is the conditional branch either looping back for another iteration or ending the loop.

2.2.3 Approaches for counting CIL instructions

To count CIL instructions, we consider two different approaches. In the following section, the two approaches we refer to as *Naive Counting* and *Counting by Interpretation*.

Naive counting is a naive approach to determine the number of CIL instructions. It works by reading all the lines of a CIL file, and counting the number of times each CIL instruction is encountered. It has the advantage of being quick, as it only needs a single pass of the code to perform the analysis. However, it does have its disadvantages. It does not account for any rules imposed by the different CIL instructions, such as branching paths in the code or looping structures. As such the naive counting approach is highly inaccurate in cases where such structures are prevalent. As an example, the code in Listing 2 contains a single *'add'* instruction in line `IL_0007`, however, because the main body of the code is a looping structure, it should be counted 100 times. Therefore for this simple example, there is a difference of two orders of magnitude between expected and actual numbers. It also does not account for the CIL instructions executed during methods calls, and as such it is only suitable for program level evaluations.

The second approach, *Counting by Interpretation*, works by attempting to simulate the CIL code, as if it were being executed. Thus, this approach interprets the code. In this approach all classes are identified and for each class their respective methods. The interpretation then starts in a specified entry method. Each CIL instruction is then simulated using a stack and heap. This approach is highly accurate, because of how close it is to actual execution. As such, loops are iterated, as many times as stated. Branching paths are determined and only the affected CIL instructions are stored in the resulting counter. A disadvantage of this approach is the speed of analysis, because the program is essentially executed line by line. Furthermore, when interpreting the code in this way, another disadvantage concerns inputs to the program as well as to methods. For the interpreter to accurately count the CIL instructions, it must know the values of inputs.

We consider the naive counting approach to be too inaccurate, as there are many programming constructs that it does not account for. Thus, we disregard this naive approach and implement the interpretation approach to count CIL instructions.

2.2.4 Static Estimation Using Machine Learning

One of the approaches that we use for static estimation is using machine learning. A similar approach is proposed in [12], however whereas their machine learning model is based on software performance features, our model predict the energy consumption based on the CIL instructions of a program. The design of such a machine learning model consists of several considerations. First, we must determine which machine learning algorithm to employ and which approach to use for evaluating the performance of the trained model. Lastly, we must consider how to create the data set for the model.

First, we consider what machine learning algorithm to employ. Machine learning is generally divided into *supervised*, *unsupervised*, *semi-supervised*, and *reinforcement* learning [21]. Supervised learning describes problems that involve using a model to learn a mapping between input examples and a target variable, while unsupervised learning describes problems that use a model to describe or extract relationships in data. Semi-supervised learning is an approach that combines both supervised and unsupervised learning. Lastly, reinforcement learning solves a problems where decision making is sequential, and the goal is long-term, such as game playing or robotics. For this project the learning is supervised, as we are interested in mapping the number of occurrences of each CIL instruction to the energy consumption of a program. Within the category of supervised learning we distinguish between *regression* and *classification* problems. Classification involves predicting a categorical variables and regression involves predicting a continuous variable. As we are interested in predicting the energy consumption, being a continuous value, we design the machine learning approach to utilise a regression type machine learning algorithm. Regression algorithms can be categorised into being either linear or non-linear. Both types are described in the following sections to outline the advantages and disadvantages of each.

Linear Regression

The *simple linear regression* algorithm models the relationship between two continuous variables, one being the *response* variable and the other being the *explanatory* variable. With linear regression we assume a linear relationship between the response and explanatory variable. When several explanatory variables are used to predict the outcome of a response variable, this is denoted as *multiple linear regression.* For this project, the energy consumption of a program, or part of a program, is the response variable, while the number of occurrences of each CIL instruction denotes the explanatory variables. This means, we have 226 explanatory variables, as there are 226 different CIL instructions. For simple linear regression, a linear regression line has an equation of the form seen in Equation (2.1) [22].

$$Y = a + bX, \tag{2.1}$$

where X is the explanatory variable and Y is the response variable. The slope of the line is b, and a is the intercept. Equation (2.2) shows multiple linear regression.

$$Y = a + b_1X_1 + ... + b_nX_n, \tag{2.2}$$

where Y is the response variable, a is the y-intercept, b_1 is the regression coefficient for the first explanatory variable (X_1), and b_n is the regression coefficient of the last explanatory variable (X_n).

To obtain predictions from linear regression the values of the regression coefficients must be learned. This is done by fitting the data to a regression line. The most common method for fitting a regression line is the method of *Ordinary Least Squares* [22]. This method computes the best-fitting line for a set of data by minimising the sum of the squares of the vertical deviations from each data point to the line. Ordinary Least Squares is given in Equation (2.3)

$$\sum_{i=1}^{M}(y_1 - \hat{y}_i)^2 = \sum_{i=1}^{M}\left(y_1 - \sum_{j=0}^{p} b_j * x_{ij}\right)^2, \tag{2.3}$$

where M is the number of instances in the data set, and p is the number of explanatory variables. Thereby, y_i is the response variable of the i'th instance in the data set. Similarly, b_j is the regression coefficient of the j'th explanatory variable, and x_{ij} is the j'th explanatory variable of the i'th instance in the data set. A common pitfall of fitting a model is that of overfitting, meaning the model does not generalise well beyond the data it has already seen. One way to combat overfitting is adding a regularisation term to the ordinary least squares equation. This approach is implemented in *Ridge Regression* and *Lasso Regression* [21]. For ridge regression the regularisation denotes a penalty, which restricts the size of the coefficients to avoiding overfitting. This amounts to the ordinary least squares with the addition of an L2 penalty term, as seen in Equation (2.4) [23].

$$\sum_{i=1}^{M}(y_1 - \hat{y}_i)^2 = \sum_{i=1}^{M}\left(y_1 - \sum_{j=0}^{p} b_j * x_{ij}\right)^2 + \lambda \sum_{j=0}^{p} b_j^2, \tag{2.4}$$

where λ is the penalty term, which regularises the coefficients such that if the coefficients take large values the optimisation function is penalised. For Lasso regression the regularisation denotes a penalty, that reduces explanatory variables that has a coefficient of zero, meaning some variables are neglected for the evaluation of the output. This is the L1 regularisation term, and the equation is seen in Equation (2.5) [23].

$$\sum_{i=1}^{M}(y_1 - \hat{y}_i)^2 = \sum_{i=1}^{M}\left(y_1 - \sum_{j=0}^{p} b_j * x_{ij}\right)^2 + \lambda \sum_{j=0}^{p} |b_j|, \tag{2.5}$$

Non-Linear Regression

Contrary to linear regression, with non-linear regression we do not assume a linear relationship between the response and the explanatory variables. Approaches to non-linear regression include *Random Forests* and *Support Vector Regression.*

Random Forest Regression operates by constructing a plethora of decision trees when training and outputting the mean prediction of all trees [21]. Random Forest Regression is thus an ensemble learning technique as it combines the predictions from multiple machine learning algorithms, in this case decision trees. Random Forest Regression uses a technique known as *bootstrap aggregation* or *bagging* [21]. This means, each decision tree is trained on a different sample of data where sampling is done with replacement. By using bagging the correlation between the decision trees is decreased, making Random Forest Regression resilient to overfitting.

Support Vector Regression is an extension to the well-known *Support Vector Machines* that allows for modelling regression tasks. Support Vector Regression is similar to Linear Regression in that it tries to fit a line, or in this case a curve, to the data. In Linear Regression the objective is to minimise the sum of squared errors, however with Support Vector Regression we can define how much error is acceptable in our model and then find an appropriate line or curve to fit the data. More specifically, instead of minimising the squared error we minimise the L2-norm of the regression coefficients, as seen in Equation (2.6) [24]

$$MIN\ \frac{1}{2}\ ||b||^2,\ where\ |y_i - bx_i| \leq \epsilon, \tag{2.6}$$

where x_i is the vector of explanatory variables with response variable y_i. Likewise b is the vector of regression coefficients. In the constraint we see the absolute error is less than or equal to a specified margin, ϵ, called the maximum error. ϵ is a hyperparameter and can be tuned to gain the desired accuracy of the model. In Support Vector Regression a *kernel* determines which kind of line or curve to fit the data to. Examples of kernels are linear, polynomial, and radial basic function (RBF). This also means, that based on the kernel, Support Vector Regression is useful for linear as well as non-linear regression tasks.

Choice and Evaluation of Models

For this project we employ an experimental approach, where we fit regressors of each of the types just described. This is to determine which algorithm suits this project best. This means, we fit five different regressors to the data set being

- Linear Regression
- Ridge Regression
- Lasso Regression
- Random Forest Regression
- Support Vector Regression

We then compare the models to determine which performs best on our problem and data. To evaluate the models we consider their performance on unseen data. This can be done in one of two ways, the first being that the data set is divided into a training and a test set. With this split, the model is fit on the training set and evaluated on the test set. The other approach is called *K-Fold Cross Validation*, with this approach the data set is divided into K folds where each fold is used as the testing set eventually. The algorithm runs in K iterations, where in the first iteration, the first fold is the testing set and the remaining $K-1$ folds are the training set. This is repeated for each fold. Then the final evaluation of the model is the mean of each iteration. K-Fold cross validation thus maximises the use of the available data for training and then testing a model. It is particularly useful for assessing model performance, as it averages a range of accuracy scores across different data sets.

Data Set Creation

For this approach the data set consists of C# programs described by the number of occurrences of each CIL instruction of each program and the program's corresponding energy consumption. It is essential to construct a large data set such that the programs collectively covers all CIL instructions. To obtain such a large data set we look for repositories of C# programs online, examples of such repositories are the *Rosetta Code*[1] repository, the *Computer Language Benchmarks Game*[2], and the benchmark suite created in our preliminary work [3]. When scraping the repositories we exclude programs that take input, either in the form of command line arguments or user input when the program is running. This is because the energy consumption of the program can be dependent on the input.

2.2.5 Static Estimation Using an Energy Model

The other approach for static estimation of a program's or function's energy consumption is by creating an energy model. This involves modelling the energy per instruction for each instruction of some low-level or intermediate-level representation of the code. In our case, we consider C#'s intermediate representation, being CIL code.

In [15] the authors propose a general framework for estimating the energy consumption of an embedded Java Virtual Machine (JVM). To do this, they perform experiments to estimate the constant overhead of the JVM energy consumption and they establish an energy consumption cost for individual Java Opcodes. This is comparable to, what we strive to achieve as the Java Virtual Machine is comparable to the .NET Common Language Runtime, and Java Opcodes are comparable to the CIL instructions. In Java as well as in C# a strict class file structure needs to be respected, it is therefore not possible to only execute one Java opcode or CIL instruction. Thus, to estimate the JVM overhead the authors create an empty Java application and measure the energy of executing this. The empty application is seen on Listing 3 on the following page. Then to estimate the energy consumption of each instruction they perform a cost comparison of Java files, for example to obtain the cost of loading an integer (iload), the energy consumption of the application seen on Listing 4 on the next page is compared to the empty application (Listing 3 on the following page) and the difference is the cost of the `iload` instruction. In this way, the authors are able to estimate the energy consumption of each Java Opcode.

The approach described above can however be simplified using the concept of *runtime code generation*, which is possible in Java as well as in C# [25]. C# natively supports runtime code generation through the classes in the `System.`-

[1]`https://www.rosettacode.org/wiki/Rosetta_Code`
[2]`https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html`

```java
public class HelloWord {
    public static void main(String arg[])
    {
        //nothing to do
    }
}
```

Listing 3: Empty Java application to measure the constant overhead

```java
public class HelloWord {
    public static void main(String arg[])
    {
        int i;
    }
}
```

Listing 4: Java application that loads an integer

`Reflection.Emit` namespace. The `System.Reflection.Emit` namespace classes can be used to emit CIL code dynamically such that the generated code can be executed directly.

In C# the `ILGenerator` is used to generate method bodies in dynamic assemblies and for standalone dynamic methods [26]. An `ILGenerator` can be obtained in two ways; either by dynamically creating an assembly, module, type, and method and then instantiating the `ILGenerator` using `MethodBuilder.GetILGenerator`. The other way is by creating the `ILGenerator` from a `DynamicMethod`. Here, instead of creating an assembly, module, type, and method, only a `DynamicMethod` is instantiated. A `DynamicMethod` represents a method that can be compiled, executed, and discarded on runtime [27].

The first approach is seen in Listing 5 on the next page, which is based on [25]. In this approach the `System.Reflection.Emit` namespace classes are used to first create an `assembly` (Line 11-13), then in that assembly create a `module` (Line 14), then a type is created in Line 15, and in that type a method is created (Line 16). Then in Line 18 the `ILGenerator` is instantiated on the method and we can emit CIL code to that generator. Lastly, in Lines 22-23 the type is actually created and the method is invoked. The other approach to invoking a method of generated CIL code is seen on Listing 6 on page 21. Here, a `DynamicMethod` is created in Line 12, and from this the `ILGenerator` is obtained in Line 13. The CIL Code is emitted to the `ILGenerator` on Lines 14-15, and the `DynamicMethod` is invoked on Line 16.

```
1  using System;
2  using System.Runtime;
3  using System.Reflection;
4  using System.Reflection.Emit;
5
6  public class Program
7  {
8      static void Main(string[] args)
9      {
10         var appDomain = AppDomain.CurrentDomain;
11         var assemblyName= new AssemblyName();
12         assemblyName.Name = "TestAsm";
13         var assemblyBuilder =
           ↪  appDomain.DefineDynamicAssembly(assemblyName,
           ↪  AssemblyBuilderAccess.Save);
14         var moduleBuilder =
           ↪  assemblyBuilder.DefineDynamicModule("TestModule");
15         var typeBuilder = mb.DefineType("TestType",
           ↪  TypeAttributes.Public);
16         var methodBuilder =
           ↪  typeBuilder.DefineMethod("TestMethod",
           ↪  MethodAttributes.Public | MethodAttributes.Static,
           ↪  null, null);
17
18         var ilg = metb.GetILGenerator();
19         ilg.EmitWriteLine("Hello World");
20         ilg.Emit(OpCodes.Ret);
21
22         var ty = typeBuilder.CreateType();
23         ty.GetMethod("TestMethod").Invoke(null, new object[] {});
24     }
25 }
```

Listing 5: Creating a method based on an assembly, module, and type

```
using System;
using System.Runtime;
using System.Reflection;
using System.Reflection.Emit;

namespace test
{
    class Program
    {
        static void Main(string[] args)
        {
            var dynamicMethod = new DynamicMethod("TestMethod",
            ↪  typeof(void), Type.EmptyTypes);
            var ilg = dynamicMethod.GetILGenerator();
            ilg.EmitWriteLine("Hello World");
            ilg.Emit(OpCodes.Ret);
            dynamicMethod.Invoke(null, Type.EmptyTypes)
        }
    }
}
```

Listing 6: Creating a method using the dynamic approach

For this project the approach to estimating the energy consumption of individual CIL instruction based on the `System.Reflection.Emit` namespace classes and the `DynamicMethod` is used. The methodology of performing cost comparisons, as proposed in [15] is also highly applicable in this case. Applications similar to Listing 3 on page 19 and Listing 4 on page 19 can be created by emitting the corresponding CIL instructions and the energy for each instruction is then computed by comparing the energy consumption of the applications.

Some instructions are data dependent, meaning they require data either in the form of arguments or on the stack. An example of such instructions are the *load* instruction which requires an argument with a value to load and the *MUL* instruction which requires two stack values to multiply. These instructions can have different energy consumption based on the values [13]. To accurately measure the energy consumption of data dependent instructions, the instruction should be provided the range of all values in its state space. However, this is practically infeasible as an instruction's state space might be infinite. Instead, for data dependent instructions, where data is provided in the form om arguments, we provide data dependent instructions with a number of random values from the instruction's state space and use these values to approximate the average energy consumption of an instruction [28].

2.3 Dynamic Estimation

In this section we decide on an approach for estimating the energy consumption of software dynamically. In Section 1.2 on page 2, we found that to perform dynamic measurements either an internal power meter, such as RAPL, or an external one, such as the WattsUP Pro, can be used. For this project using an external power meter is practically infeasible. This is because we create an IDE extension for developers to use, we thus cannot assume the developer to have an external device connected nor can we control such an external device remotely. Thus, we decide to implement our dynamic measuring approach using the internal power meter, RAPL. In this section we first elaborate on the capabilities, pros, and cons of RAPL, after which we elaborate on how we can tailor the use of RAPL to the needs of our IDE extension.

2.3.1 Measuring Framework using RAPL

Intel introduced the on-chip power meter Running Average Power Limit (RAPL) in their CPU architecture since the introduction of the Sandy Bridge line of processors. Furthermore, RAPL allows for a high temporal resolution of a millisecond, and a spatial resolution of a set of power planes including the power consumption of the entire CPU or DRAM [17]. A more detailed description of RAPL is found in our preliminary research [3] and in [17].

One approach for estimating the energy consumption of a software system is to collect readings from RAPL while executing the entire program. This has the advantages of being very accurate as the reading reflect the energy consumption of the CPU or DRAM and are measured directly on the CPU. Disadvantages for this approach is the noise from the rest of the system. As RAPL not only measures the energy consumption of the running program, but also from the rest of the system. Therefore, background tasks and the operating system itself are also reflected in the readings. Furthermore, at the time of writing, the RAPL readings are only available on the Linux operating system. Another disadvantage for this approach is that the entire program is tested at once. This could make it more difficult to compare different algorithms or measure a subset of code. Furthermore, if the code that is to be measured depends on other parts of the program, then both parts are measured at the same time, which can make it difficult to determine where the energy consumption lies.

In Section 2.1 on page 5 we decide that energy consumption measurements must be available for entire programs as well as individual methods. Thus, we must implement some functionality to account for this. To do this, we build upon the `measuring` framework of our preliminary research, found in [3]. In our preliminary work, we create a framework that provides statistically significant energy consumption measurements of code. This framework uses RAPL for measuring and the results are statistically significant as Cochran's formula is used to compute the number of executions of the code to perform. Our framework can obtain measurements for all power planes of the CPU, however for this project we only consider the package power, as this is where most energy is spent. Our extension to this framework consists in creating custom attributes that let the developer annotate their code with what code to measure the energy. The design of this framework draws inspiration from unit-testing frameworks. Thus, developers will create a new class annotated with a custom attribute letting the framework know, this class is relevant. Within the class methods can be created within which the developers can instantiate and execute relevant classes and methods of their code, for which their energy consumption will be measured. These measurement methods will be annotated with custom attributes as well, telling the framework to perform measurements on the code within. Similar to unit-testing frameworks, we also allow for set-up and tear-down methods where the code from these is executed before and after the execution of the measurement methods respectively. A proposed design of this is seen on Listing 7 on the next page

This method has the advantages of only measuring a subset of the code, and input can be ignored from the measurements, by residing within the constructor or a setup class. However, since this method also relies on RAPL, the disadvantages regarding RAPL still persists, such as handling noise and it only be available on Linux systems with an Intel CPU.

```csharp
namespace MeasureTesting
{
    [MeasureClass]
    public class MeasurementClass
    {
        [MeasureSetup]
        public void Setup()
        {
            /* Setup for the measure methods */
        }

        [Measure]
        public void MeasureMethod()
        {
            /* Any functionality to be measured */
        }
    }
}
```

Listing 7: An example of a measure framework based on a standard unit testing framework

2.4 Extension

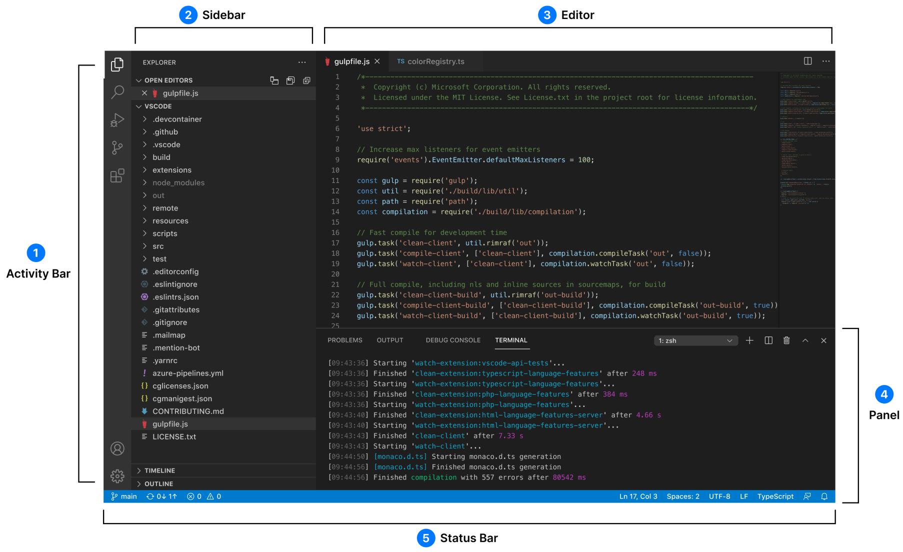
This section covers the design of the IDE extension for Visual Studio Code. First, an introduction to extensions in Visual Studio Code is covered, followed by the initial design of the user interface (UI) of the extension.

2.4.1 IDE introduction

IDE extensions serve to provide new features for an existing IDE. An extension can provide support for developing programs in a new language, or adding snippets to ease the load for the developer.

The goal for this project is to aid developers in reasoning about the energy consumption of their code. For this we create an IDE extension for Visual Studio Code. Extensions can contribute to Visual Studio Code in multiple ways, including: [29]

- Change the look of VS Code with a colour or file icon theme
- Add custom components & views in the UI
- Create a Webview to display a custom webpage built with HTML/CSS/JS



```
/*---------------------------------------------------------------------------------------------
 *  Copyright (c) Microsoft Corporation. All rights reserved.
 *  Licensed under the MIT License. See License.txt in the project root for license information.
 *--------------------------------------------------------------------------------------------*/

'use strict';

// Increase max listeners for event emitters
require('events').EventEmitter.defaultMaxListeners = 100;

const gulp = require('gulp');
const util = require('./build/lib/util');
const path = require('path');
const compilation = require('./build/lib/compilation');

// Fast compile for development time
gulp.task('clean-client', util.rimraf('out'));
gulp.task('compile-client', ['clean-client'], compilation.compileTask('out', false));
gulp.task('watch-client', ['clean-client'], compilation.watchTask('out', false));

// Full compile, including nls and inline sources in sourcemaps, for build
gulp.task('clean-client-build', util.rimraf('out-build'));
gulp.task('compile-client-build', ['clean-client-build'], compilation.compileTask('out-build', true))
gulp.task('watch-client-build', ['clean-client-build'], compilation.watchTask('out-build', true));
```

```
[09:43:36] Starting 'watch-extension:vscode-api-tests'...
[09:43:36] Finished 'clean-extension:typescript-language-features' after 248 ms
[09:43:36] Starting 'watch-extension:typescript-language-features'...
[09:43:36] Finished 'clean-extension:php-language-features' after 384 ms
[09:43:36] Starting 'watch-extension:php-language-features'...
[09:43:40] Finished 'clean-extension:html-language-features-server' after 4.66 s
[09:43:40] Starting 'watch-extension:html-language-features-server'...
[09:43:43] Finished 'clean-client' after 7.33 s
[09:43:43] Starting 'watch-client'...
[09:44:50] [monaco.d.ts] Starting monaco.d.ts generation
[09:44:56] [monaco.d.ts] Finished monaco.d.ts generation
[09:44:56] Finished compilation with 557 errors after 80542 ms
```

Figure 2.4.1: Architecture of Visual Studio code [29]

- Support a new programming language - Language Extensions Overview
- Support debugging a specific runtime

Section 2.2 on page 8 and Section 2.3 on page 22 described different approaches for estimating the energy consumption of a program. Since the estimation techniques have different shortcomings and advantages, the different approaches are collected into a single extension, which gives the user a selection of options to chose from. For the user to get dynamic estimates they must be using a Linux machine where RAPL can be accessed (see Section 2.3 on page 22, however the static measurements are available for all users.

2.4.2 Design of User Interface

In Visual Studio Code, there exists different view groups that an extension can influence, see Figure 2.4.1. The activity bar (1) provides a menu for extensions which uses the sidebar, where the sidebar (2) functions as the view of the extension. Therefore, we design an extension which provides an icon for the activity bar, along with a view for the side bar.

The user interface (UI) of the extension for the sidebar panel needs to contain a way of choosing which estimation approach to use, be it static or dynamic, along

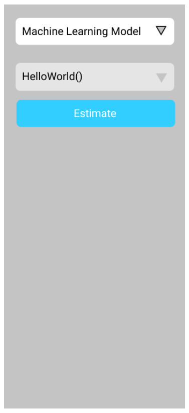


Figure 2.4.2: An initial design for the User Interface for the extension

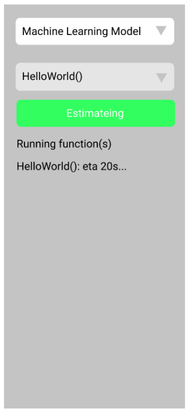


Figure 2.4.3: An initial design for the User Interface for the extension while estimating

with a way of describing which code to estimate. Furthermore, the extension should report the result of the estimation to the user. This can be done in several ways: by saving the result to a file, showing the result directly in the extension, open a new window with the results, or possibly annotate the code directly with the estimation results.

Based on the above requirements for the UI, an initial design is created which is seen on Figure 2.4.2. The initial design contains a drop-down menu for selecting the estimation approach. Below this, another drop-down menu shows each method in the currently open program. This drop-down allows the user to choose individual methods to perform estimations on. They can also choose to measure on the entire program. Finally, below this an *estimate* button is present, which starts the estimation approach for the selected method. While the estimation is running, a different UI is shown. This is seen in Figure 2.4.3 where the button has changed colour, and the progress of the running methods is shown below.

As for the results from the estimation, we choose to save the results to a file on the host system, as well as to show them directly on the extension when the estimation is done.

Chapter 3

Implementation

In this section the implementation of the estimation techniques and the IDE extension is elaborated on. First, in Section 3.1 we outline the general architecture of our implementation. Then in Section 3.2 on page 30 we elaborate of the implementation of our static estimation techniques and in Section 3.3 on page 46 we detail the dynamic measurement approach.

3.1 Architecture

The IDE extension consists of a frontend for the user to interact with, and a backend consisting of microservices that compute the energy estimations using the various techniques described in Chapter 2 on page 5. A figure of the general architecture is seen on Figure 3.1.1 on the following page

The frontend is implemented as a `WebView` (see Section 3.4 on page 51) using HTML, CSS and JavaScript. When the user requests an energy estimate, the extension sends this request to the appropriate microservice. The microservices, `Measure Service`, `Machine Learning Service`, and `Energy Modelling Service`, represent the three implemented approaches to estimation. The `Measure Service` provides dynamic energy estimates by executing the code and using RAPL to measure. The `Machine Learning Service` and the `Energy Modelling Service` provide the static estimates. For this, both services rely on the `Instruction Counter` microservice. This microservice is implemented as an interpreter for

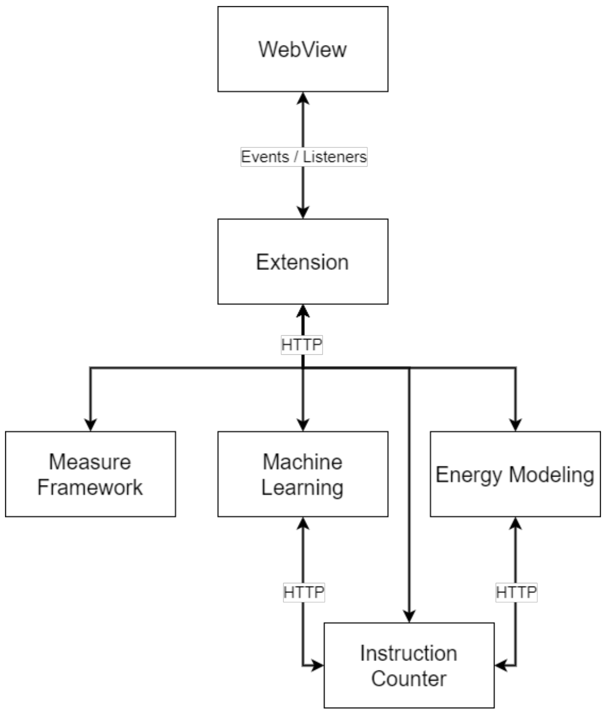


Figure 3.1.1: Simple overview of the architecture of the IDE extension

CIL code that, given an assembly file, counts the number of times each CIL instruction is executed. The `Machine Learning Service` uses the instruction counts as input to the regression models when predicting the energy consumption. Furthermore, the `Energy Modelling Service` uses the `Instruction Counter` microservice to multiply the energy consumption of each instruction with the number of times, they occur. All services are elaborated on in the following sections.

3.1.1 API

The frontend of the IDE extension communicates with the microservices using REST. The API is seen in Table 3.1.1 on the following page where all endpoints are compiled.

Table 3.1.1: API endpoints for the different microservices

Method	**Microservice**	**Address**
POST	Measure Service	/estimate
GET	Measure Service	/progress
POST	Measure Service	/stop
GET	Measure Service	/methods
POST	Machine Learning Service	/estimate
POST	Energy Model Service	/estimate
POST	Instruction Counter Service	/counts

The microservices `Measure Service`, `Machine Learning Service`, and `Energy Model Service` each expose an endpoint for getting an energy consumption estimate. These are the `/estimate` endpoints. To accomplish this the frontend provides the microservices with an array of objects of type `ActivateClass`, see Listing 8 or IDs in the case of dynamic estimation. This array of `ActivateClasses` identifies the methods, and their containing classes. For each class a field denotes where its assembly file is located. This information is essential as the `Instruction Counter` library relies on the assembly file to count instructions.

```
export interface ActivateClass {
    ClassName: string;
    AssemblyPath: string;
    Methods: Method[];
}

export interface Method {
    Id: number;
    Name: string;
    StringRepresentation: string
}
```

Listing 8: A snippet showing which types of data the `ActivateClass` contains

In addition to endpoints allowing the user to get energy estimates, it also exposes endpoints for getting information about the progress of the ongoing measurement and for stopping the ongoing measurement. The dynamic approach needs to run the code of each function multiple times to get a statistically significant results, whereas the static approaches are only run one time for any given estimate. Therefore, we only provide the progress endpoints for the dynamic approach. The last two endpoints, `/methods` and `/counts` are utility methods.

The former being an endpoint for getting all of the methods for a given assembly file. The latter, an endpoint to count all of the CIL instructions for a given assembly file or for individual methods in that assembly.

3.2 Static Estimation

In this section the static estimation approaches are elaborated on. As mentioned in Section 2.2 on page 8 and Section 3.1 on page 27 both approaches rely on being able to precisely determine which CIL instructions are executed and how many times. Thus, we elaborate on the implementation of how to obtain precise instruction counts. Then we detail each of the approaches for obtaining static energy measurements. We consider the estimation approach using machine learning. We elaborate on, how the machine learning models are trained and we describe the implementation of the microservice that provide energy estimates using the machine learning models. Likewise, for the static estimation approach using an energy model we describe how we create the energy model and how the microservice for the energy model is implemented.

3.2.1 CIL Instruction Counting

In this section, we present the implementation of our interpretation approach as described in Section 2.2.3 on page 12 that we use for counting CIL instructions. Before being able to interpret a program some preprocessing first be executed. This preprocessing is presented in Figure 3.2.1 on the next page. From the top, the instruction counter program can take either an assembly (.DLL) file, where the CIL code is extracted using a utility called `ILspycmd`, or it can be given the CIL code directly. The CIL code is then parsed based on the grammar from the ECMA standards 335 [30]. The result from the parser is a list of classes and methods, where classes contain their parent classes, and all the methods belonging to it. After the classes and methods are created, we can being interpreting the code. In the following sections we first detail our implementation of the IL parser component of Figure 3.2.1 on the following page after which we elaborate on how we implement the actual interpretation.

Parser

The CIL code is run through a parser to create objects, which can be manipulated as part of the counting process. The three most important result objects

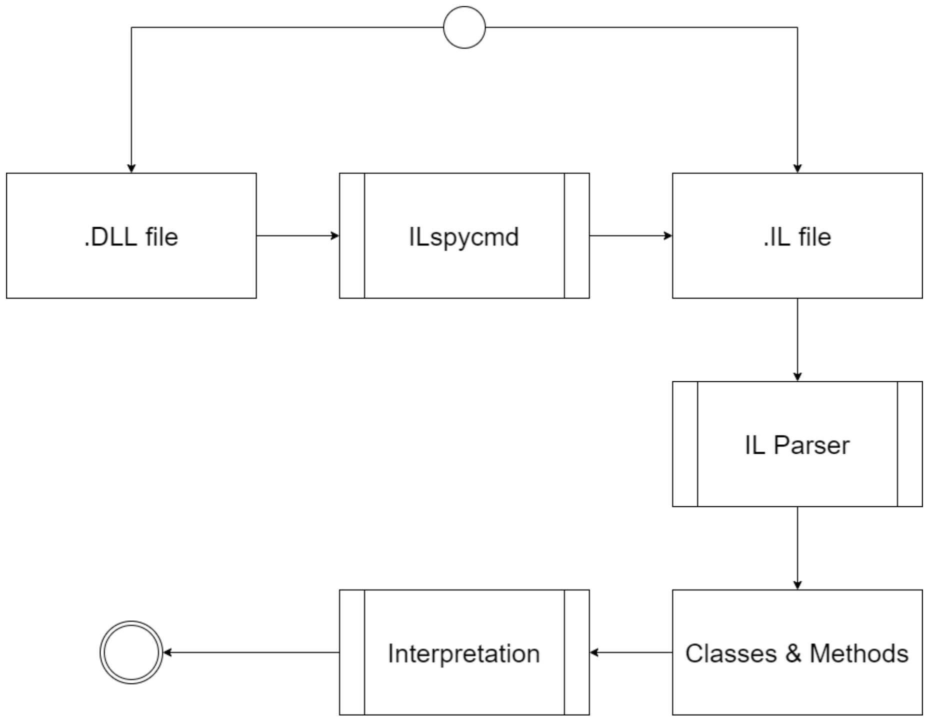


Figure 3.2.1: A simple overview of the overall structure for counting instructions

are the `Class`, `Method`, and `Instruction`. The instruction is the lowest level of information, and also the most important part of the CIL code, since these are the elements we want to count. The simple representation of an instruction can be seen on Listing 9 on the next page. It contains three pieces of information: The `location`, which is a hex value for the location of the instruction in a method. The `name`, which is the actual name of the instruction, such as, *nop* or *brfalse*. Lastly the `data` is everything after the name, which describes all the necessary parameters for any instruction which require such parameters. An example of this division of data can be seen below.

$$\overbrace{IL\_000c}^{Location}: \underbrace{brtrue.s}_{Name} \overbrace{IL\_0005}^{Data} \tag{3.1}$$

The next level out from the instruction, is the method level. A method object is helpful to keep track of which instructions are mapped to which method names.

```python
class Instruction():
    def __init__(self, location, data):
        self.location = 'IL_' + location
        name, *data = data.split()
        self.name = name
        self.data = data
```

Listing 9: The simple representation of an instruction

The information for a method can be seen on Listing 10, which shows a reduced subset of the information contained within. Two interesting parameters are `is_entry` and `instructions`. Both parameters are set later, since a method header is parsed before the internals are parsed. The `is_entry` field, is set if the keyword `.entrypoint` is found within a method. This signifies that the specific method is the starting point when executing the program. The other value `instructions`, is a hash map used to get all instructions for the given method. These instructions are instances of the instruction object above.

```python
class Method():
    def __init__(self, method_attr, call_conv, return_type,
    ↪   marshal, method_name, gens, params):
        self.attributes = method_attr
        self.call_conversions = call_conv
        self.generics = self.load_generics(gens)
        self.is_generic = bool(gens) == True
        self.is_entry = False
        self.is_instance = 'instance' in call_conv
        self.parameters = params
        self.instructions = {}
```

Listing 10: A simplified view of the data for a method

The outermost level is the class, which is necessary, since multiple methods can have the same name and the same parameters, but be part of different classes. Therefore, to know the exact method for counting instructions, we have to provide the class name, along with the method name. Combining all three levels of information, we can find a specific method and get all of the instructions contained within.

Counting by Interpretation

The approach to counting instructions that we implement is the `Counting by Interpretation` approach. The continuation of the process named `interpretation` on Figure 3.2.1 on page 31, can be seen on Figure 3.2.2. This approach requires either that the program has an `entrypoint` or that a starting method is specified. All programs that are meant to be run standalone, must contain an entrypoint method. The process starts by finding either the entrypoint or the specified method. All of the instructions for the method are loaded onto an instruction stack, and all the necessary initialisation is done: heap, stack, static storage. Then the interpretation is started. Each instruction is evaluated in turn. If an instruction is a call instruction, the call is made and the control-flow is changed to the other method. This process is run until we evaluate the return instruction from the initial method. Whenever an instruction is encountered, it is recorded in a set of executed instructions. This set is created per method, and upon calling the return instruction for a given method, the set is added to a global set of counters. At the very last step, when the process has finished interpreting, all the intermediate counters are summarised into a single final count of instructions.

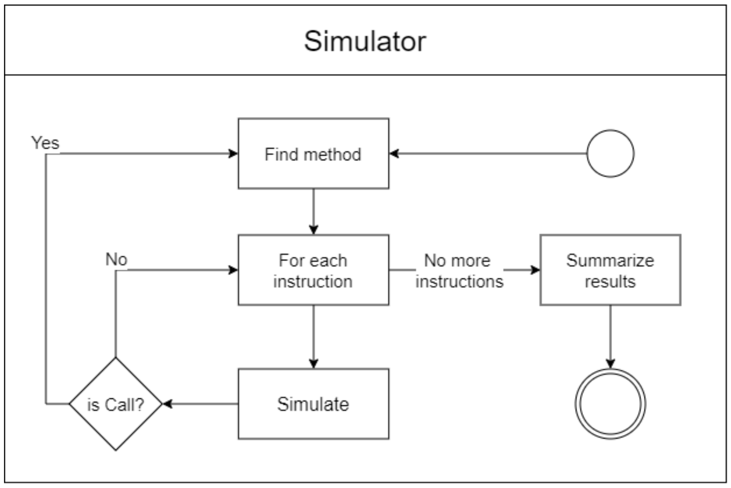


Figure 3.2.2: An overview of the steps for the counting by interpretation approach

One of the key methods in the interpreter is the `execute_method` method, which takes a method and loads all of the contained instructions into a list. It then iterates through the instructions until either a return statement is evaluated, or it runs out of instructions. The code for this method can be found on Listing 11 on the next page. Lines 2 through 5 is the setup, where instructions are loaded and initial values are set. In the loop, the instruction is interpreted on Line 9 by calling the `execute` method on the instruction. This method has two return values, first is the action to perform after evaluating the instruction. For example if the instruction is a branch instruction, then the next action is to `JUMP` to some other instruction in the list of instructions. The other return value contains additional information in regards to the next action. In the example with the branch instruction, the second return value might be the new location, which is the target of the branch instruction. There are four types of actions available in the executor. `JUMP`, Line 11, is for changing control-flow to another instruction in the same method. `CALL`, Line 13, is when another method has to be executed, this is the case for regular method calls, but also for constructors when creating new objects. `NOP`, Line 21, is the most common action, and signifies that no additional action has to take place, before the next instruction can be executed. The last action type: `RETURN`, Line 23, is used, when the method is returning.

We create the interpreter with the intention of handling all C# programs. However, due to the time constraints of this project, there are however a set of cases, for which the interpreter is unable to perform the interpretation of the code. These limitations are discussed in Chapter 5 on page 70.

3.2.2 Static Estimation Using Machine Learning

In this section, we describe the implementation of the static estimation approach using machine learning. First, we elaborate on how the machine learning models are created and fit, and then we detail the implementation of the machine learning microservice, which the frontend extension interacts with. The machine learning models as well as the microservice are implemented in Python, as Python provides a plethora of machine learning libraries which suits the purpose of this approach.

```python
def execute_method(self, method):
    instructions = method.get_instructions()
    instruction_index = list(instructions.keys())
    index = 0
    return_val = None

    while index != len(instruction_index):
        current = instruction_index[index]
        action, value =
        ↪  instructions[current].execute(self.storage)

        if action == Actions.JUMP:
            index = instruction_index.index(value)
        elif action == Actions.CALL:
            machine = state_machine(self.storage)
            return_val = machine.simulate(value,
            ↪  self.storage.get_active_class())
            value.clear()
            if return_val or return_val == 0:
                self.storage.push_stack(return_val)
            self.storage.pop_active_class()
            index += 1
        elif action == Actions.NOP:
            index += 1
        elif action == Actions.RETURN:
            return_val = value
            break

    return return_val
```

Listing 11: The executor method for the interpreter

As described in Section 2.2.4 on page 13 the machine learning approach consists of several steps. The individual steps are compiled in the list below and each of the steps are elaborated on in the following text.

1. Create data set by scraping C# programs from online repositories
2. Count CIL instructions for each program in the data set
3. Measure the energy consumption for each program in the data set
4. Fit regression models to the data

The first point is to create a data set by discovering and scraping C# programs from online repositories. For this, we have scraped the *Rosetta Code*[1] repository and the benchmark programs from *Computer Language Benchmarks Game*[2]. Furthermore, the learning platforms of *Sanfoundry Global Education and Learning* [3] and *Include Help* [4] contains a plethora of C# programs, which are also included in the data set. Lastly, we include all benchmark programs created in our preliminary work presented in [3]. We scrape these websites and our preliminary work for C# programs and end up with 1438 programs for the data set. However, not all programs scraped are included in the final data set. This is because we exclude programs if they conform to one of the following conditions:

1. Does not have a `Main` function
2. Waits for user input
3. Requires command line arguments
4. Cannot be build without modifications to the code or the setup
5. Cannot be run with our interpreter

If a program does not have a `Main` function, the program cannot be executed without modifications. It is essential that the programs can be executed, otherwise the energy consumption of the program cannot be measured. If the program waits for user input the energy cannot be reliably measured, because it will depend on what input is given and how quickly. Furthermore, if the program requires input in the form of command line arguments we discard the program as well. This is because, we cannot infer what input is required, and the energy consumption will be dependent on the given input. In addition, in our preliminary work of [3], we show that IO is very energy intensive, and thus including IO will overshadow the energy consumption of the rest of the function being measured. We also discard programs that cannot be build without modifying the code or setup. This is generally due to non-standard libraries being used, for example from NuGet packages. Lastly, we exclude programs that our interpreter cannot handle. This is because the interpreter counts the CIL instructions of each program, which eventually are the explanatory variables of the training set.

[1]https://www.rosettacode.org/wiki/Rosetta_Code
[2]https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html
[3]https://www.sanfoundry.com/csharp-programming-examples/
[4]https://www.includehelp.com/dot-net/basic-programs-in-c-sharp.aspx

In total, we end up with 147 programs for the training set. The distribution of the programs is seen on Figure 3.2.3. This chart shows the percentage of programs that are excluded due to the reasons mentioned above. The `remaining` portion shows the percentage of programs that are not excluded (10.2 %). As can be seen the largest contributors to exclusion of programs are that they do not contain a `Main` function (22.2 %) or that our interpreter cannot handle the program (35.9 %). For the pie-chart it is worth mentioning, that if a program fails several criteria it is only present in *one* category. The order of priority follows the order with which the criteria are presented in the enumerated list above.

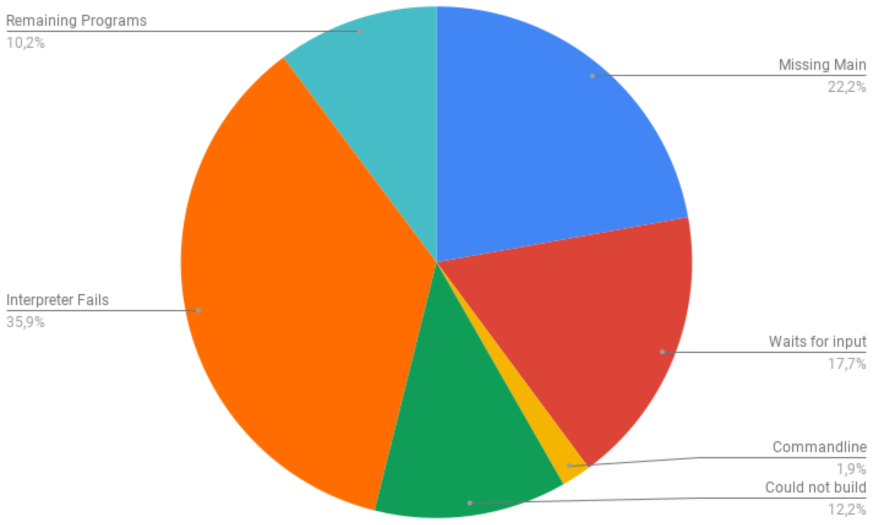


Figure 3.2.3: Distribution of scraped programs.

Having scraped the C# programs, the data set is created by counting the CIL instructions of each program using our interpreter and the approach described in the previous section. We measure the energy consumption of each program using the approach and framework presented in our preliminary research [3]. The final data set is presented as a csv file, and a snippet of the data set is seen on Table 3.2.1 on the next page.

Having created the data set, the regressors are fit. Listing 12 on the following page shows a snippet from the Python script that fits the regressors. In the script `X` denotes the vector of explanatory variables and `y` is the response variable. The

Table 3.2.1: Number of instructions for different benchmarks

Name	*pkg(μj)*	*add*	*add.ovf.un*	*and*	*arglist*	...	*unbox*	*unbox.any*	*xor*
example-of-a-...	34154.90	0	0	0	0	...	0	0	0
find-sum-of-all-...	36220.16	327	0	0	0	...	0	0	0
print-the-integer-...	36299.79	0	0	0	0	...	0	0	0
csharp-program-...	35339.00	0	0	0	0	...	0	0	0
Combinations_2	71289.25	0	0	0	0	...	0	0	0

script takes as an input parameter what kind of regressor to fit, and based on that, the method `get_model` called on Line 46 instantiates a regressor of that type in the `model` variable. On Line 49 the model is fit to the data set. The model is cross validated on Lines 50-52 if an input argument specifies to do so. Likewise, the input argument also controls whether to save the model for future use (Lines 54-55).

```
43      X = pd.DataFrame(df.drop(['name', 'pkg(μj)', 'duration(ms)',
         ↪  'dram(μj)', 'temp(C)'], axis=1))
44      y = pd.DataFrame(df['pkg(μj)'])
45
46      model = get_model(args.regression_method)
47      y = np.ravel(y)
48
49      model.fit(X, y)
50      if args.cross_validate:
51          splits = int(len(y) /10)
52          print(np.mean(cross_val_score(model, X, y,
             ↪  cv=splits,scoring='neg_root_mean_squared_error')))
53
54      if args.save_model:
55          pickle.dump(model, open('model.obj','wb'))
```

Listing 12: Snippet of script to train a regression model

Machine Learning Microservice

The microservice that provides the frontend with energy estimates is written in Python as an aiohttp[31] application. This allows the frontend and the microservice to communicate using HTTP requests. A snippet from the implementation is seen on listing 13 on the next page

As mentioned in Section 3.1 on page 27 the endpoint receives an array of type `ActivateClass`, thus in the microservice implementation we iterate over each class to make predictions on the given methods in each class. First, we retrieve

```python
    activate_classes = json_data['activeClasses']
    inputs = json_data['inputs']
    all_predictions = {}
    for current_class in activate_classes:
        path_to_assembly = current_class['AssemblyPath']
        className = current_class['ClassName']
        methods = current_class['Methods']
        abs_file_path = os.path.splitext(path_to_assembly)[0]
        name = os.path.split(abs_file_path)[-1]

        # dissassemble and get il code
        subprocess.call(f'ilspycmd {path_to_assembly} -o . -il',
        ↪  shell=True)
        text = open(f'{name}.il').read()

        # count instructions, maps method/program name to IL
        ↪  instruction Counter
        counts = requests.post('http://localhost:5004/counts',
        ↪  json={'path_to_assembly' : path_to_assembly,
        ↪  'methods': methods, 'inputs': inputs, 'class_name':
        ↪  className})
        counts = counts.json()

        # make prediction
        predictions = {} # maps method/program name to energy
        ↪  prediction
        model = pickle.load(open(model_path, "rb"))
        with open('CIL_Instructions.txt') as f:
            CIL_INSTRUCTIONS = [x.strip() for x in f.readlines()]

        for name, count in counts.items():
            count = reduce(lambda a, b: Counter(a) + Counter(b),
            ↪  count, count[0])
            temp = []
            for instruction in CIL_INSTRUCTIONS:
                temp.append(count[instruction]) if instruction in
                ↪  count else temp.append(0)
            predictions[name] = model.predict([temp])[0][0] /
            ↪  1000000
        all_predictions[className] = predictions

    # return result
    return web.Response(text=json.dumps(all_predictions),
    ↪  status=200)
```

Listing 13: Snippet of machine learning microservice

the assembly path, the name of the class, and the array of methods on which we will make predictions. Then the assembly file is disassembled using `ilspycmd` on Line 35 and the resulting `.il` file is read on Line 36. On Line 39 our Instruction Counter library is used to obtain the number of times each instruction occur in each of the methods. Then beginning from Line 43 the energy estimates are predicted. First, the trained machine learning model is loaded on Line 44 and on Line 45 a list of all CIL instructions is loaded. The loop beginning on Line 48 iterates through all methods in the current class that we want to obtain energy estimates from and their corresponding CIL counts. Thus, `name` is the name of the method and `count` represents the instruction counts. The `count` variable is a list of type `Counter` mapping CIL instructions to their number of occurrences. It is a list of `Counters`, since, if the method calls other methods a new `Counter` is instantiated to count the CIL instructions executed in that method. On Line 49, the list of `Counters` are reduced to one `Counter` in the variable `count`, and then for each CIL instruction, if that instruction is present in `count` its count is appended to the `temp` list, otherwise a `0` is appended, denoting the instruction has not been executed. Then, using the `temp` array, the `model` predicts the energy consumption of the given method on Line 53, and the resulting energy value is divided by 1,000,000 to convert micro Joule (μJ) to Joule (J). We convert from micro Joule to Joule for readability as the energy consumption of the majority of the programs corresponds to Joule or millijoules and they are thus simpler to compare when converting to Joules. The prediction is inserted into the `predictions` dictionary, and when estimates are computed for all methods in a given class, the `predictions` dictionary is inserted into the `all_predictions` dictionary on Line 67. Lastly, the `all_predictions` dictionary is returned to the frontend as the response to the HTTP request.

3.2.3 Static Estimation Using an Energy Model

Based on the description in Section 2.2.5 on page 18, an energy model is created for each CIL instruction using the dynamic RAPL framework approach initially described in Section 2.3 on page 22 and the implementation is introduced in Section 3.3.1 on page 47. This framework is used to simplify the creation of the energy model, and to be responsible for obtaining statistically significant measurements.

The following description relates to the method seen on Listing 14 on the following page. The first step in creating an energy model for CIL instructions, is to create a DynamicMethod where the instructions are emitted to, along with an ILGenerator. Therefore, a method called NewMethod is created which returns a new DynamicMethod and its corresponding ILGenerator. This method is seen in Listing 14 on Lines 4-9, where a DynamicMethod is instantiated with the name MyMethod, return type of void, and which takes no arguments to run. Furthermore, for the measurement of each instruction, the method needs to make a return call and the dynamic method should be *invoked*.

```
4      private (DynamicMethod, ILGenerator) newMethod()
5      {
6          DynamicMethod method = new DynamicMethod("MyMethod",
           ↪  typeof(void), new Type[] { });
7          var ilg = method.GetILGenerator();
8          return (method, ilg);
9      }
10
11     private void runMethod(DynamicMethod method, ILGenerator ilg)
12     {
13         ilg.Emit(OpCodes.Ret);
14         method.Invoke(null, Type.EmptyTypes);
15     }
```

Listing 14: Creating and invocation of DynamicMethod

Before creating the benchmarks for measuring the energy consumption of each instruction, we first measure the constant overhead of creating and executing a DynamicMethod with no other CIL instructions emitted to it than OpCodes.Ret (emitted in runMethod). The method for this is seen in Listing 15 on the following page on Lines 18-22. As can be seen both methods in Listing 15 on the next page has the Measure attribute, which is necessary for the measuring framework to know which methods to benchmark. For the Empty method the attribute specifies how many executions to use for its the pilot run. The other method seen on Listing 15 on the following page is for computing the energy consumption of the instruction Ldc_I4, which loads a 32-bit integer onto the stack. In this case the Measure attribute also takes as input an array denoting which instructions the execution of the current instruction depend on. Thus, the energy consumption costs for the dependencies can be subtracted to the cost of the current instruction, to achieve the actual value. In the case of Ldc_I4 it depends on no other instructions being executed beforehand, thus the array given as input to the Measure attribute only consists of Empty, denoting that only the constant overhead should be subtracted.

When emitting the Ldc_I4 instruction it is required to also provide the integer value to load onto the stack. The stream of OpCodes emitted to the ILGenerator must represent a valid program, which means that the stack must be empty before returning, as the DynamicMethod has a void return type. Therefore, the OpCode Pop is emitted to the generator as well, which then renders the method as valid. However, this has the implication that when run, we measure the energy consumption of not only the Ldc_I4 instruction, but also a Pop instruction. As the DynamicMethod we create has a void return type, no program can be made which only loads values onto the stack without also popping the stack. If instead the DynamicMethod had a return type a value is allowed on the stack when returning. However, for simplicity, we do not return any values.

```
17      [Measure(1000)]
18      public void Empty()
19      {
20          var (method, ilg) = newMethod();
21          runMethod(method, ilg);
22      }
23
24      [Measure(10000, new []{ "Empty" })]
25      public void Ldc_I4(int value)
26      {
27          var (method, ilg) = newMethod();
28          ilg.Emit(OpCodes.Ldc_I4, value);
29          ilg.Emit(OpCodes.Pop);
30          runMethod(method, ilg);
31      }
```

Listing 15: An example of measuring the energy consumption for the constant overhead, and the energy consumption of the CIL instruction Ldc_I4

Like the Ldc_I4 instruction (see Listing 15 on Lines 25-31), some instructions require some value to be emitted with the instruction. These are the data dependent instructions. To provide these instructions with some data the framework searches for any input parameters a method may take and supplies random values of this type. The implementation of this is seen in Listing 16 on the following page. The energy consumption of data dependent instructions correlate with the data itself ([13, 28]), thus we use random values instead of fixed values to ensure that the results can be more generalised. When using random values we achieve an average of the cost of the instruction. Alternatively, we could provide the instruction with the entire state space of possible values and then average the energy consumption. However, in practice, this is infeasible as the state space of an instruction can be large or even infinite [13]. When measuring the energy consumption of an instruction using the measurement framework, the method representing the instruction is executed multiple times to ensure statistical significance of the energy consumption value obtained. Each time the method is executed it is provided a new random value.

```
object[] randomInputs = method.GetParameters().Select(parameter => {
    var rnd = new Random();
    var typeSwitch = new Dictionary<Type, Object> {
        { typeof(int), rnd.Next(int.MinValue, int.MaxValue) },
        { typeof(uint), ((uint)rnd.Next(int.MinValue, int.MaxValue) + (uint)int.MaxValue) },
        { typeof(short), (short)rnd.Next(short.MinValue, short.MaxValue) },
        { typeof(ushort), ((ushort)rnd.Next(ushort.MinValue, ushort.MaxValue) + (ushort)ushort.MaxValue) },
        { typeof(sbyte), (sbyte)rnd.Next(-128, 127) },
        { typeof(byte), (byte)rnd.Next(0, 255) },
        { typeof(long), (long)rnd.Next(int.MinValue, int.MaxValue) },
        { typeof(float), (float)rnd.NextDouble() },
        { typeof(double), rnd.NextDouble() },
        { typeof(string[]), new string[]{ "one", "two", "three" } },
        { typeof(string), new string(Enumerable.Repeat("ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789", rnd.Next(1, 1000) )
                .Select(s => s[rnd.Next(s.Length)]).ToArray())},
        { typeof(bool), rnd.Next(0, 1) },
        { typeof(Type), allTypes[rnd.Next(0,allTypes.Length-1)]},
        { typeof(PosInt), new PosInt() {i = rnd.Next(1, Int16.MaxValue)}}
    };
```

Listing 16: Creating a random value based on a type

When all instructions have had their energy consumption measured, we need to remove any dependencies from the results. This includes, removing the overhead of the empty model from each instruction model. Dependencies denote any instruction that must be executed prior to the instruction in question, for example, the `Brtrue` instruction requires an integer value on the top of the stack, which is asserted on as the branch condition. Therefore, to measure the energy consumption of the `Brtrue` instruction, an integer is loaded onto the stack. Thus, unlike `Ldc_I4`, the `Brtrue` instruction has multiple dependencies, namely `Empty` and `Ldc_I4`. Therefore, in addition to the constant overhead denoted by the `Empty` method, the energy consumption value of `Ldc_I4` must also be subtracted to get the measurements of only the `Brtrue` instruction.

Once all dependencies are subtracted from the energy consumption values of each instruction, an output XML file is created containing all methods and costs (see listing 17 on the next page). This XML file is then used with the CIL instruction counter presented in section 3.2.1 on page 30, to compute the energy consumption of a program or method.

For the energy model we implement 175 of 226 instructions. We elaborated more on this in Chapter 5 on page 70.

Energy Model Microservice

The energy model microservice runs in the background allowing the frontend to make HTTP requests to get the energy consumption using the energy model estimation technique. Similar to the machine learning microservice, this microservice is written in Python using aiohttp to supports HTTP requests.

The Energy Model microservice requires an XML-file representing the energy consumption of each instruction, as shown in listing 17 on the following page. This model is precomputed and thus available for the microservice to use. However, as different CPUs may vary in energy usage per instruction the results only reflect the energy consumption of the CPU on which the energy model (and thus XML-file) is created. Though not supported, it would be beneficial to allow the user to create their own energy model, reflecting the energy consumption using their CPU.

Listing 18 on page 46 shows a snippet of the implementation of the energy model microservice. The energy model microservice exposes a single endpoint which expects an array of type `ActivateClass` (see Section 3.2.2 on page 38). Similar to the machine learning microservice this array is iterated beginning

```xml
<class>
  <name>measureClass</name>
  <method>
    <declaring-type>Modeling.measureClass</declaring-type>
    <name>Empty</name>
    <result>Passed</result>
    <measurement>
      <name>timer</name>
      <mean>0,04722468999999934</mean>
      <completed-runs>10000</completed-runs>
      <deviation>0,0216284220475747</deviation>
      <ErrorMargin>0,000216284220475747</ErrorMargin>
      <ErrorPercent>0,00457989709357O1224</ErrorPercent>
      <mean-subtracted>0,04722468999999934</mean-subtracted>
    </measurement>
  </method>
  <method>
    <declaring-type>Modeling.measureClass</declaring-type>
    <name>Ldc_I4</name>
    <result>Passed</result>
    <measurement>
      <name>timer</name>
      <mean>0,04744672999999965</mean>
      <completed-runs>10000</completed-runs>
      <deviation>0,016627105080843554</deviation>
      <ErrorMargin>0,00016627105080843554</ErrorMargin>
      <ErrorPercent>0,0035043732372797187</ErrorPercent>
      <mean-subtracted>0,000222040000000312</mean-subtracted>
    </measurement>
    <dependencies>
      <instruction>Empty</instruction>
    </dependencies>
  </method>
</class>
```

Listing 17: A snippet of the energy model

from Line 62. Then the assembly file is disassembled using `ilspycmd` and the .il file is loaded and given as input to the Instruction Counter library. Then from Line 78 each method and their respective instruction count is iterated. For each method all instructions that are executed in that method are iterated and multiplied by the energy consumption for that instruction based on the energy model XML file. Lastly, the collected energy for the instructions are returned to the frontend.

```
    # Read the request info
    fileinfo = await request.json()
    activate_classes = json_data['activeClasses']
    all_results = {}
    for current_class in activate_classes:
        path_to_assembly = current_class['AssemblyPath']
        class_name = current_class['ClassName']
        methods = current_class['Methods']
        abs_file_path = os.path.splitext(path_to_assembly)[0]
        name = os.path.split(abs_file_path)[-1]

        # dissassemble and get il code
        subprocess.call(f'ilspycmd {path_to_assembly} -o . -il',
        ↪  shell=True)
        text = open(f'{name}.il').read()

        # Count instructions
        counts = get_cil_counts(methods, class_name)

        # Calculate measurements for all methods in class
        results = {}
        for method_name, counter in counts.items():
            counter = reduce(lambda a, b: a+b, counter,
            ↪  counter[0])
            sum = 0.0
            for instruction in counter:
                count = counter[instruction]
                instruction = ILToEmit(instruction)
                if instruction in ILModelDict:
                    cost = ILModelDict[instruction]
                    sum += count*cost
            results[method_name] = sum
        all_results[class_name] = results

    # return result
    return web.Response(text=json.dumps(all_results), status=200)
```

Listing 18: Snippet of the energy model microservice

3.3 Dynamic Estimation

This section covers the implementation of the dynamic energy consumption estimation approach presented in Section 2.3 on page 22. The dynamic estimation approach executes the program or method, while measuring the energy using

RAPL, and immediately returns the result to the frontend. In the following section this entire approach is elaborated on.

3.3.1 Measuring Framework

This section covers the creation of a measurement framework for C# programs, which utilises and extends our previous work presented in [3]. We adhere to the design presented in Section 2.3 on page 22. This means, we create a framework that allows the user to annotate classes and methods with our custom attributes, that denote on which methods the energy consumption should be measured. Thus, we extend our preliminary work of [3] with the ability of recognising custom attributes. To create a custom attribute in C# we follow the guidelines presented in [32] and first create a class that inherits from `Attribute`. The input variables to the attribute corresponds to the inputs to the constructor of the custom attribute class. We can annotate our custom attribute class with other attributes to further specialise the uses of the custom attribute. This includes which targets the attribute applies to, such as assembly, class, method, field and more. Furthermore, we can define whether the attribute is allowed to be applied multiple times to the same target.

In this project, we create four custom attributes `MeasureAttribute`, `Measure-ClassAttribute`, `MeasureSetupAttribute` and `MeasureCleanupAttribute`. All but the `MeasureClassAttribute` applies to methods, where the class attribute applies to classes. Furthermore, none of the custom attributes are allowed to be applied multiple times to the same target. The implementation of the `Measure-Attribute` is seen in listing 19. The class constructor takes the number of sample iterations to be performed, and stores this in the class along with the total planned iterations, which is updated in a later step. Moreover, the measure attribute class contains a list of all completed measurements for this method.

The framework is extended with the functionality of detecting and handling the custom attributes. Next, the process of obtaining the energy consumption of the given program or method, and returning the results to the extension frontend, is broken down into several steps. Each step in the process is seen in the list below and elaborated on in the following text.

1. Detect classes implementing the `MeasureClassAttribute`

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
public class MeasureAttribute : Attribute
{
    public List<Measurement> Measurements;
    public int SampleIterations;
    public int PlannedIterations;
    public int IterationsDone;

    public MeasureAttribute(int sampleIterations = 100)
    {
        SampleIterations = sampleIterations;
        PlannedIterations = sampleIterations;
        Measurements = new List<Measurement>();
    }

    public void AddMeasure(Measure measure)
    {
        this.IterationsDone++;
        foreach (var api in measure.apis)
        {
            if (Measurements.Any(m =>
            ↪  m.Name.Equals(api.apiName)))
            {
                Measurements.First(m =>
                ↪  m.Name.Equals(api.apiName)).AddMeasurement(api.apiValue);
            }
            else
            {
                var temp = new Measurement(api.apiName);
                temp.AddMeasurement(api.apiValue);
                Measurements.Add(temp);
            }
        }
    }
}
```

Listing 19: Implementation of the MeasureAttribute

2. Detect any methods, within a class implementing the `MeasureClassAttribute`, which implements either `MeasureAttribute`, `MeasureSetupAttribute` or `MeasureCleanupAttribute`

3. Execute the setup method, the measurement method(s) and the cleanup method until we have a statistically significant result.

As the user should be allowed to select a set of methods to run, these three steps are broken up into two separate endpoints on the *MeasureFramework* microservice. The first endpoint (*GetMethods*) implements the two first steps, and provides the IDE with a set of classes and methods which contains the custom attribute along with an ID of the methods. The second endpoint (*Estimate*) takes a set of IDs from the IDE and run only the specified methods along with any setup and/or cleanup methods found in the class of the method.

GetMethods

First, to detect the classes that implement the attribute `MeasureClassAttribute`, the assembly file of the project is loaded into an assembly using the `System.Reflection` namespace. From this assembly class, we iterate over all types, and collect the types which implement the attribute `MeasureClassAttribute`.

With a class type, we can extract all attributes from the class, in this way we can extract our custom attributes and their corresponding fields and methods. Furthermore, an object of type `Type` contains a method called `GetMethods` which returns an array of methods of type `MethodInfo`, the methods can then be filtered such that only methods containing a `MeasureAttribute`, `MeasureCleanupAttribute` or `MeasureSetupAttribute` remains. The methods are then given an ID which in this case is the hash of that method and are saved on the microservice. The methods are then returned to the IDE. This can all be seen in Listing 20 on the next page

Estimate

The IDE provides a set of IDs for the methods which should be tested. The methods are then found in the array, generated by the `GetMethods` endpoint. Thus, a reference to the class, class-attribute, method and method-attribute is obtained along with a reference to the setup and cleanup methods. The setup method is then invoked on the class, performing any setup as specified in that method, on that class. The method is then invoked on the class using the `measurement` framework from [3] such that all measurements from the framework is saved in the `MeasureAttribute`. This is done in a loop for the number of sample iterations specified by the user. Following the guidelines from [3], we use Cochran's formula to calculate the iterations needed to obtain a statistically significant result, and perform any extra iterations of the method. Lastly, the cleanup method is invoked on the class.

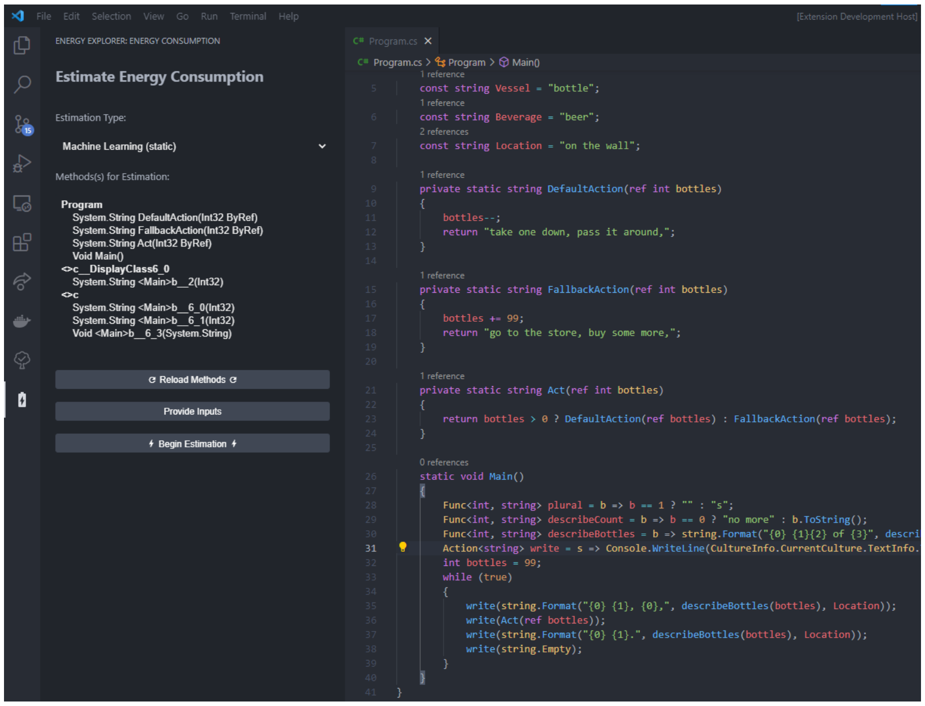
```
private List<ClassMethods> getAllMethods(Type currentClass, string file, bool getWithAttributes)
{
    List<ClassMethods> result = new List<ClassMethods>();
    MethodInfo[] allMethods = currentClass.GetMethods(BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Static | BindingFlags.Instance).Where(mi => mi.DeclaringType == currentClass).ToArray();

    if (getWithAttributes)
        allMethods = allMethods.Where(m => m.GetCustomAttributes().Any(a => a is MeasureAttribute)).ToArray();

    if (allMethods.Any())
    {
        MethodViewModel[] methodViewModels = allMethods
                .Select(m => new MethodViewModel()
                    {
                        Id = m.GetHashCode(),
                        Name = m.Name,
                        Args = m.GetParameters().Select(p => p.ParameterType.ToString()).ToArray(),
                        StringRepresentation = m.ToString()
                    })
                .ToArray();
        ClassMethods cm = new ClassMethods
        {
            CurrentClass = currentClass,
            AssemblyPath = file,
            Methods = methodViewModels,
        };
        result.Add(cm);
        if (getWithAttributes)
            Methods.Add(cm);
    }
    return result;
}
```

Listing 20: Shows how to get all methods of a class

As the energy consumption of all methods are measured, we create an XML file containing a list of each class and its methods. For each method, we include the number of executions performed, along with all the types of measurements performed and the statistically significant results in the XML file. Lastly, the XML file is returned to the IDE frontend.

3.4 Extension

Extensions in Visual Studio Code are created as *TreeViews* or *WebViews*. TreeViews denote interfaces similar to a file explorer, where content is structured as a tree and conforms to the style of the built-in views of Visual Studio Code. On the contrary, WebViews allow for the creation of fully customisable views using HTML, CSS, and JavaScript to create the user interface. To conform to the design presented in Section 2.4 on page 24, we thus create a WebView. The final user interface is seen on figure 3.4.1.


File
Edit
Selection
View
Go
Run
Terminal
Help
[Extension Development Host]
ENERGY EXPLORER: ENERGY CONSUMPTION
Program.cs
Program.cs > Program > Main()
Reload Methods
Provide Inputs
Begin Estimation

Estimate Energy Consumption

Estimation Type: Machine Learning (static)

Methods(s) for Estimation:

```
Program
   System.String DefaultAction(Int32 ByRef)
   System.String FallbackAction(Int32 ByRef)
   System.String Act(Int32 ByRef)
   Void Main()
<>c__DisplayClass6_0
   System.String <Main>b__2(Int32)
<>c
   System.String <Main>b__6_0(Int32)
   System.String <Main>b__6_1(Int32)
   Void <Main>b__6_3(System.String)
```

```
    1 reference
    const string Vessel = "bottle";
    1 reference
    const string Beverage = "beer";
    2 references
    const string Location = "on the wall";

    1 reference
    private static string DefaultAction(ref int bottles)
    {
        bottles--;
        return "take one down, pass it around,";
    }

    1 reference
    private static string FallbackAction(ref int bottles)
    {
        bottles += 99;
        return "go to the store, buy some more,";
    }

    1 reference
    private static string Act(ref int bottles)
    {
        return bottles > 0 ? DefaultAction(ref bottles) : FallbackAction(ref bottles);
    }

    0 references
    static void Main()
    {
        Func<int, string> plural = b => b == 1 ? "" : "s";
        Func<int, string> describeCount = b => b == 0 ? "no more" : b.ToString();
        Func<int, string> describeBottles = b => string.Format("{0} {1}{2} of {3}", descri
        Action<string> write = s => Console.WriteLine(CultureInfo.CurrentCulture.TextInfo.
        int bottles = 99;
        while (true)
        {
            write(string.Format("{0} {1}, {0},", describeBottles(bottles), Location));
            write(Act(ref bottles));
            write(string.Format("{0} {1}.", describeBottles(bottles), Location));
            write(string.Empty);
        }
    }
}
```

Figure 3.4.1: A view of the extension

As can be seen, the extension allows the user to select which type of estimation technique they want to use, along with the methods for which they want to obtain energy consumption estimates. In the bottom of the view three buttons allow the user to refresh the methods shown, provide inputs for the methods, and begin the estimation.

To create the Visual Studio Code extension we create a `WebView` and a `WebView Provider`. The `WebView` contains the UI code, being the HTML, CSS and JavaScript to manipulate the UI. The `WebView provider` represents the back-end code that is responsible for communicating with the various microservices. The `WebView` and the `WebView Provider` communicate by posting messages to each other. A message is a dictionary and consists of a `type` and a `value`. This allows us to differentiate between the calls, and parse the message to the correct type in either the `WebView` or the `WebView Provider`. In the `WebView` as well as in the `WebView Provider` a switch case uses the type as the condition and determines which actions to perform when receiving messages. An example of the `WebView` posting a message is seen on Listing 21, and likewise an example of the `WebView Provider` posting a message is seen on Listing 22. The switch-case of the `WebView Provider` is seen on Listing 23 on the following page, where it can be seen, that the `WebView Provider` can handle messages of the types: `log`, `activate`, `stop`, and `reloadMethods`. Likewise, the `WebView` handles messages of the types: `progress`, `done` and `methods`, where for each of these, the value of the message contains objects or information to update the UI with.

```
webviewView.webview.postMessage({ command: 'done', value:
↪  response });
```

Listing 21: `WebView` posting a message

```
vscode.postMessage({
    type: "activate",
    value: { methods: selected, inputs: inputsDict, type: type },
});
```

Listing 22: `WebView Provider` posting a message

For the `WebView Provider` to handle the messages posted by the `WebView` it must be able to communicate with the various microservices. As can be seen in Listing 23 on the next page all messages are handled in the `MeasureParser` class. In this class methods are created to parse the messages and communicate with the microservices. An example of this is seen on Listing 24 on page 54. The `MeasureParser` then awaits a response from the server, and parses the response to the `WebView`.

```
webviewView.webview.onDidReceiveMessage(message => {
  switch(message.type){
    case 'log':
      console.log(message.value);
      break;
    case 'activate':
      let methods = message.value.methods as ActivateClass[];
      let type = message.value.type;
      let inputs = message.value.inputs
      Measure.activate(methods, inputs, type, webviewView);
      break;
    case 'stop':
      Measure.stop();
    case 'methodSelected':
      break;
    case 'reloadMethods':
      Measure.getMethods(webviewView, message.value.type);
      break;
    default:
      console.log("Cound not understand message of type: " +
      ↪  message.type);
      break;
  }
});
```

Listing 23: Switch-case of `WebView Provider`

```
static callService(url: string, httpMethod: any, data: any =
↪  null): Promise<any> {
    return new Promise(async (resolve, reject) => {
        try {
            let response;
            if (data != null) { response = await httpMethod(url,
            ↪  data); }
            else { response = await httpMethod(url); }

            if (response.status == 200) {
            ↪  resolve(response.data); }
            else { reject(response.status); }
        } catch (exception) {
            reject(exception);
        }
    });

}
```

Listing 24: `WebView Provider` communicating with microservices

Chapter 4

Experiments and Results

We have developed an IDE extension for Visual Studio Code with the purpose of helping developers reason about the energy consumption of their code. The extension contains three approaches to estimate the energy consumption of C# programs and methods. These are: dynamically using RAPL and statically using either machine learning or an energy model. Furthermore, we test five different regression techniques for the machine learning approach to determine which suits the problem of this project best.

In this section we describe experiments to test the accuracy of each of our estimation techniques. In addition to testing the accuracy, we also measure the time consumption of obtaining the energy consumption estimates dynamically and using both static approaches. We use the RAPL measurements as the ground truth for the experiments. First, in Section 4.1 we describe our hypothesis regarding how each technique performs. Then in Section 4.2 on page 57 we detail the approach with which the results are obtained. In Section 4.3 on page 64 we describe the setup for the experiments, which involves the hardware and software. Lastly, in Section 4.4 on page 64 we present the results.

4.1 Hypothesis

Based on the related work presented in Section 1.2 on page 2, the choices made in Chapter 2 on page 5, and the different approaches to estimating energy consumption, we have created three hypotheses. The first is about our expectations for the performance difference for the machine learning models. The second hypothesis is about the performance of the energy model we have created. The final hypothesis concerns the execution speed of the static estimation approaches compared to the dynamic estimation approach.

Performance of machine learning models

We have five different machine learning models to test. While three of the models assume the data to be linear, we also include the random forest regression and support-vector regression, which are both non-linear. As presented in Section 1.2 on page 2 energy models typically depend on more factors than merely the number of times each instruction is executed. These factors include the energy consumed by the transition between instruction and inter-instruction effects such as cache misses and pipeline stalls. These factors point to the energy model not being linear based on the instruction count. Thus, based on these observations we expect the non-linear machine learning models to provide better predictions than the linear models.

Performance of the energy model

We have created an energy model to estimate the energy consumption based on the count of each instruction, encountered during the static analysis. It does not contain the energy consumption for all of the available instructions, and therefore we predict that the model underestimates the actual value. Furthermore, the energy model only takes into account the number of each instruction, and not the relationship between the instructions. The relationship in this case being which instructions come before others, such that we can compute the energy consumption of the transition between instructions. Thus, we anticipate that the energy model will further underestimate the energy consumption of a given program.

Execution time for static estimates and dynamic estimates

The biggest distinction between measurement approaches is the difference between static estimation and dynamic estimation. While we expect the dynamic estimation approach to obtain the actual energy consumption, we expect the static estimation approaches to just approximate the actual consumption. Using the dynamic approach, we obtain statistically significant results, compared with the static approach where the predictions are estimates. This means, for the dynamic approach, each benchmark is executed multiple times, to be able to reason about the significance. This is in contrast to the static approaches, which only have to run once. Based on this, we expect the static estimation approaches to be at least an order of magnitude faster compared to the dynamic approach.

4.2 Measurement Approach

This section covers the approach with which the experiments are conducted. We first elaborate on the suite of benchmarks that is used for the experiments. We describe how the actual energy and time consumption values are obtained for the collected benchmarks. We also detail how the energy and time consumption is estimated using the static estimation techniques. For our experiments we use the `measurement` framework and the `benchmark` library we created in our preliminary work [3]. Thus, when using the terms `measurement` framework and `benchmark` library, the libraries presented in our preliminary work are what we refer to.

4.2.1 Benchmark Suite Creation

The benchmarks, which are used for the experiments, are the same benchmarks which are used as the data set for the machine learning models. This dataset is described in Section 2.2.4 on page 13 and in Section 3.2.2 on page 34. A scraper is created to scrape a set of websites for C# programs, which in total found 1438 programs. However, when filtering the programs based on the criteria listed in Section 2.2.4 on page 13, only 147 benchmarks remain which can be used for the machine learning and energy model estimation techniques.

The selection of 147 programs is then build and run to obtain the ground truth. This is done using the `measurement` framework, which ensures that all benchmarks are run a specified number of times such that the statistical error is below the desired value. This number is computed using Cochran's formula. This process was executed on the same setup as the energy models were created on.

4.2.2 Obtaining the Energy Results

To obtain the results of each estimation technique we first compute the ground truth values. The ground truth values denote the energy consumption for each benchmark. These are computed such that we can compare the results from the static estimation techniques to the ground truth and reason about the error of each technique. To obtain the ground truth values, we use the `measurement` framework, which ensures that the energy consumption values are statistically significant. This is done by first performing a simple random sample, that is, running the benchmark for a predefined number of times (in this case 1000), to be able to compute the standard deviation for that sample. Using the values computed from the simple random sample, the Cochran formula [33] is used to determine the minimum number of runs required for the results to be within the desired error. Then, the benchmark is executed that number of times to obtain statistically significant results. For a more detailed description of the approach and the Cochran formula see [3]. For the benchmarks to be used with the `measurement` framework, each benchmark must implement the `Benchmark` library. Thus, we create a script to automatically insert the code for the `benchmark` library into each of the benchmarks. An example of a benchmark before and after the library is implemented is seen on Listing 25 on the next page and Listing 26 on the following page. As the energy consumption of all benchmarks is computed, the `measurement` framework creates files with the values for the package power, DRAM power and run time. Each file consists of values for the following columns: name, Sample Mean, Standard Deviation, Standard Error, Standard Error (%), Number of Runs. In our case, we are only interested in the sample mean value of the package power Section 2.3 on page 22.

To obtain the energy consumption values for the benchmark using the machine learning and energy model approaches, another script is created. This script requires a *csv* file as a training set for the machine learning models and an *xml* file representing the energy model. The script follows the following outline:

1. Load all benchmarks
2. Load the machine learning training set and the energy model
3. For each benchmark
    (a) Count the CIL instructions

```
using System;

class Program {
    static void Main(string[] args) {
        for (int i = 1; i <= 10; i++) {
            Console.Write(i);

            if (i % 5 == 0) {
                Console.WriteLine();
                continue;
            }

            Console.Write(", ");
        }
    }
}
```

Listing 25: Example of benchmark without the `Benchmark` library implemented.

```
using System;
using Benchmark;

class Program {
    static void Main(string[] args) {
        var bm = new Benchmark(1);
        bm.Run(() => {
            for (int i = 1; i <= 10; i++) {
                Console.Write(i);

                if (i % 5 == 0) {
                    Console.WriteLine();
                    continue;
                }

                Console.Write(", ");
            }

        return "Continue_0";});
    }
}
```

Listing 26: Example of benchmark with the `Benchmark` library implemented.

(b) Fit each of the regressors to the training set (excluding the row in the training set denoting the current benchmark)

   (c) Predict the energy consumption of the benchmark with each of the regressors

   (d) Compute the energy consumption of the benchmark using the energy model

   (e) Save the result

4. Add the ground truth values to the result and save to a csv file

The implementation of this is seen on Listing 27 on page 62 and an excerpt of the final output with all results is seen in Table 4.2.1 on the next page. In Table 4.2.1 on the following page all values are rounded to two decimals and the names are shortened for brevity.

Table 4.2.1: Excerpt of the final energy results. All values are presented in µj and rounded to two decimals. For brevity the column names are shortned, such that `GT` means `Ground Truth`, `RF` means `Random Forest`, and `EM` means `Energy Model`

Name	***GT***	***Linear***	***Lasso***	***Ridge***	***RF***	***SVR***	***EM***
example-of-a-...	34154.90	50841.86	49958.22	35529.66	34615.61	36716.96	1048.76
find-sum-of-all-...	36220.16	-10725989.13	-3589324.82	65549.08	37463.33	36700.30	321881.15
print-the-integer-...	36299.79	5601014691.34	43635.96	64684.71	36418.81	36697.64	3969.75
csharp-program-...	35339.00	29403.54	35214.19	40969.91	33443.83	35649.32	82.51
Combinations_2	71289.25	50569.10	56236.08	71685.74	44539.97	36714.50	1372.33

4.2.3 Obtaining the Time Consumption Results

To obtain the results for the time consumption of the various estimation techniques, we use the same script as described above and seen on Listing 27 on the next page. This script we modify, such that, we also measure the time it takes to count the CIL instructions and obtain the machine learning and energy model results respectively. When measuring the time, we execute each 1000 times and compute the mean value. This is alternatively to obtaining statistically significant results as, for the energy consumption, using Cochran's formula. However, in this case it is not essential for the results to be statistically significant. This is because, we are only interested in determining roughly the difference in time for obtaining statistically significant energy consumption measurements and for estimating them statically. Listing 28 on page 63 shows a snippet of the same script as seen in Listing 27 on the next page, but modified to also measure time. As can be seen the parts of the code we want to measure is executed 1000 times and then the mean of those 1000 executions is saved. Lastly, an excerpt of the final output is seen on Table 4.2.2 on page 64

```python
if __name__ == "__main__":
    # Read all benchmarkable benchmarks
    benchmarks = os.listdir('MLApproach/correct_benchmarks')
    energy_df = pd.DataFrame()

    (...)

    ITERATIONS = 1000

    for benchmark in benchmarks:
        path = f'/MLApproach/correct_benchmarks/{benchmark}/bin/Debug/net5.0/project.dll'
        energy_results = {'name' : benchmark}
        # count instructions, maps program name to IL instruction Counter
        counts = requests.post('http://localhost:5004/counts', json={'path_to_assembly' : path, 'methods': None, 'class_name': "hey", 'inputs': None})
        counts = counts.json().get('project')
        counts = reduce(lambda a, b: Counter(a) + Counter(b), counts, counts[0])

        # Run with ML
        for model in regressions:
            new_df = df[df.name != benchmark]
            X = pd.DataFrame(new_df.drop(['name', 'sample mean (µj)'], axis=1))
            y = pd.DataFrame(new_df['sample mean (µj)'])
            y = np.ravel(y)
            model.fit(X, y)
            energy_results[str(model).split('(')[0]] = get_ml_result(counts, model, CIL_INSTRUCTIONS)

        # Run with energy model
        energy_results['energy-model'] = get_energy_model_result(counts, ILModelDict, benchmark)

        energy_df = energy_df.append(energy_results, ignore_index=True)
```

Listing 27: Snippet of the script that computes the energy consumption using the static estimation techniques.

```python
if __name__ == "__main__":
    # Read all benchmarkable benchmarks
    benchmarks = os.listdir('MLApproach/correct_benchmarks')
    energy_df = pd.DataFrame()
    time_df = pd.DataFrame()

    (...)

    ITERATIONS = 1000

    for benchmark in benchmarks:
        time_measurements = []
        path =
        ↪  f'/MLApproach/correct_benchmarks/{benchmark}/bin/Debug/net5.0/project.dll'
        energy_results = {'name' : benchmark}
        time_results = {'name' : benchmark}
        # count instructions, maps program name to IL instruction
        ↪  Counter
        for _ in range(ITERATIONS):
            start = time.time()
            counts =
            ↪  requests.post('http://localhost:5004/counts',
            ↪  json={'path_to_assembly' : path, 'methods': None,
            ↪  'class_name': "hey", 'inputs': None})
            end = time.time()
            time_measurements.append((end-start)*1000)
        time_results['counts (ms)'] = np.mean(time_measurements)
        time_measurements = []
        counts = counts.json().get('project')
        counts = reduce(lambda a, b: Counter(a) + Counter(b),
        ↪  counts, counts[0])

        (...)

        # Append results
        time_df = time_df.append(time_results, ignore_index=True)
        energy_df = energy_df.append(energy_results,
        ↪  ignore_index=True)
```

Listing 28: Snippet of the script that measures the time consumption while computing the energy consumption using the static estimation techniques.

Table 4.2.2: Excerpt of the final time consumption results. All values are presented in ms and rounded to two decimals. For brevity the column names are shortened, such that `RF` means `Random Forest`, and `EM` means `Energy Model`

Name	***Counts***	***Linear***	***Lasso***	***Ridge***	***RF***	***SVR***	***EM***	***Dynamic***
example-of-a-...	272.00	0.12	0.13	0.20	4.87	0.09	0.05	78536.80
find-sum-of-all-...	287.54	0.12	0.17	0.37	5.00	0.90	0.60	36117.18
print-the-integer-...	410.30	0.12	0.12	0.25	5.00	0.09	0.07	35710.60
csharp-program-...	244.73	0,12	0.13	0.10	5.00	0.10	0.04	3640.00
Combinations_2	322.13	0.12	0.13	0.15	4.84	0.09	0.05	44901.60

4.3 Test Setup

To get results from the different estimation approaches which are compared, we run all experiments on the same desktop PC. The specifications of the PC can be seen in Table 4.3.1. This is the same desktop PC which we used in our previous work [3]. Thus, it is running a Skylake processor from Intel which enables us to use RAPL in order to get accurate energy measurements for the benchmarks. For the operating system, we have upgraded to Ubuntu version 20.04 LTS. Furthermore, we have also upgraded the .NET runtime environment to .NET 5, as it is the latest official release at the time of writing.

Table 4.3.1: Hardware specifications of test setup

General	
Processor	Intel Core i7-6700K Skylake 4 GHz Quad-Core
Storage	240GB SSD
Memory	8GB RAM
Operating System	Ubuntu 20.04 LTS
Runtime	.NET 5

4.4 Results

As described in the Section 2.2 on page 8, we have two types of static estimation: using machine learning models, and using an energy model. For the machine learning models, we test a total of five different regression techniques. Thus, we have a total of six different approaches to compare for the results. For the experiments we collect energy consumption and time consumption using the `measurement` framework of [3], and we also estimate the energy and measure the time of using both of the static approaches. The approach for this is described in Section 4.2 on page 57. The results are all present in the documents and files attached to this report, under the names *time_results.csv* and *energy_results.csv*.

In this section we present the results and compare the ground truth energy consumption values to the energy consumption estimated by both static approaches. We also compare the five different regression models among themselves to determine which suits the problem of this project the best. Lastly, we investigate the time efficiency of obtaining the static results compared to obtaining the ground truth. All of the results are gathered and presented in this section.

4.4.1 Energy Estimation Results

First, we consider the results obtained using the various machine learning techniques. Then we consider those obtained using the energy model. We compare the results to the ground truth values.

Machine Learning

Of the five different machine learning models, three boxplots are created based on the percentage difference from the ground truth and the actual estimation values which can be seen in Section 4.2.2 on page 58. These three boxplots are of random forest, support-vector regression (SVR), and lasso regression. The last two machine learning approaches are not represented because the size of the outliers make it practically impossible to display the boxplots, such that the quartiles can be viewed. The first set of boxplots (Figure 4.4.1 on the next page) contains the non-linear models random-forest and support vector regression which are the two most comparable. Furthermore, although the random forest has more outliers, it does have a tighter set of quartiles, where the minimum value is -7.5% and the maximum value is 9.2%. While for the support-vector regression boxplot, the minimum and maximum are approximately -28% and 17% respectively.

The other boxplot contains the Lasso regression results (Figure 4.4.2a on page 68). A large difference between this plot and the previous plots is the change in scale. Here many of the outliers are ranged between -500% and 500%. Though the minimum and maximum of the boxplot are ranged between -51% to 70%.

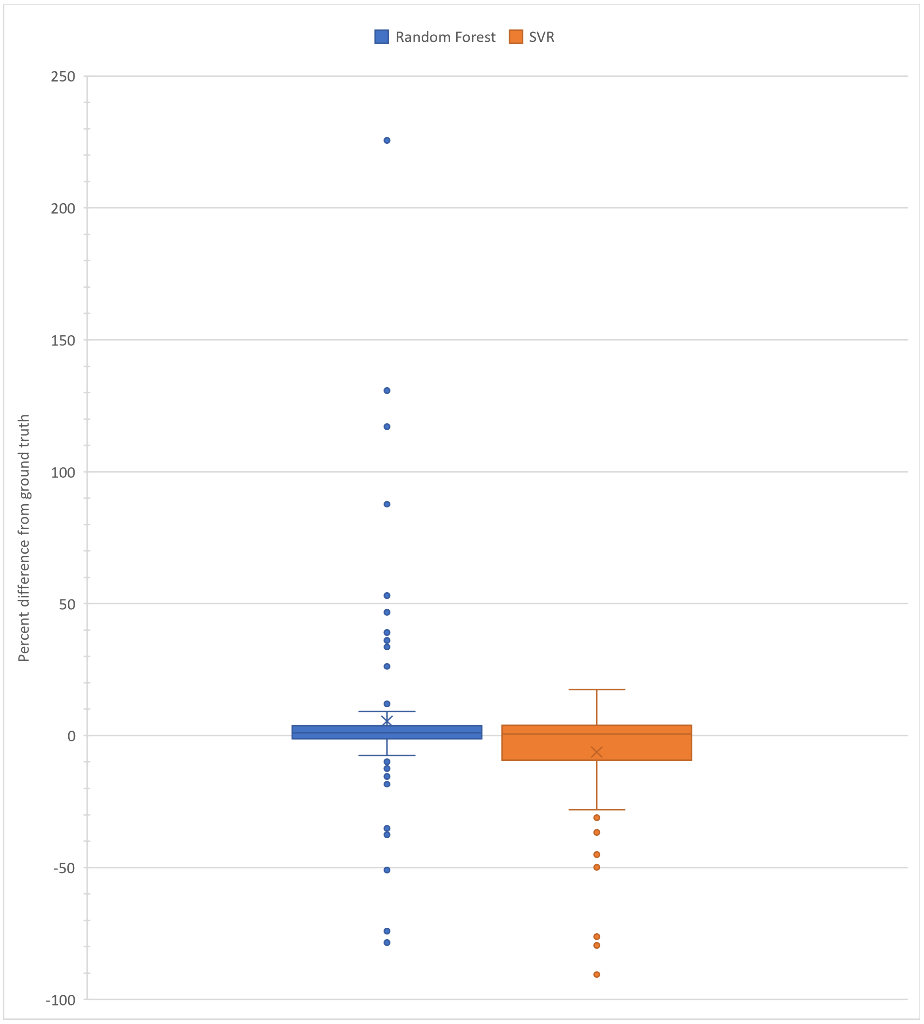


Figure 4.4.1: Boxplots for Random-Forest and Support-Vector regression

For the last two machine learning approaches, instead of providing a graphical boxplot, the quartile ranges are displayed in Table 4.4.1. To compare all of the approaches the table contains the quartiles for all of the approaches. They are ordered from lowest percentage deviation to the highest percentage deviation.

Table 4.4.1: Boxplot quartiles for the five machine learning approaches

Name	Min	Q1	Median	Q3	Max
Random Forest	-7.49%	-1.28%	1.06%	3.72%	9.19%
Support Vector	-28.06%	-9.33%	0.56%	3.83%	17.39%
Lasso	-51%	-3.67%	26.15%	36.92%	70.07%
Ridge	-219.68%	-59.56%	13.37%	68.26%	206.84%
Linear	-240.81%	-62.69%	22.49%	88.33%	302.43%

Energy Model

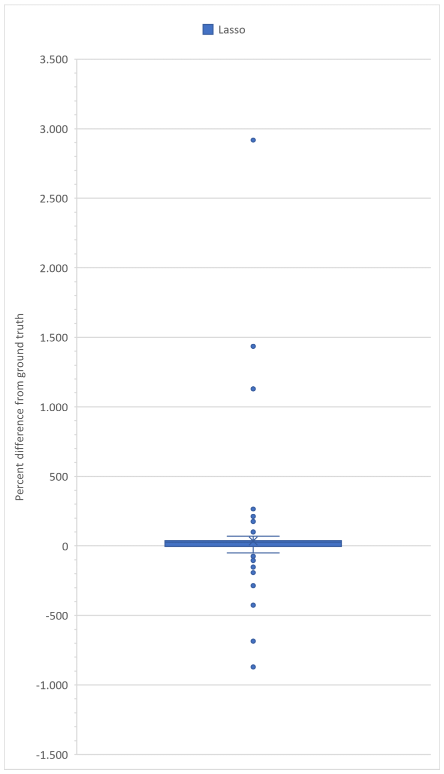
In addition to the machine learning models, we have also tested the energy model. The results of this model can be seen on Figure 4.4.2b on the following page, which is a boxplot of the results. It should be noted that the boxplot has more outliers than is shown, the x-axis has been reduced to better show the actual boxplot. Below can be seen the written form of the boxplot for easier comparison with the other results.

Name	Min	Q1	Median	Q3	Max
Energy Model	-100.56%	-91.9%	-79.22%	-21.98%	74.98%

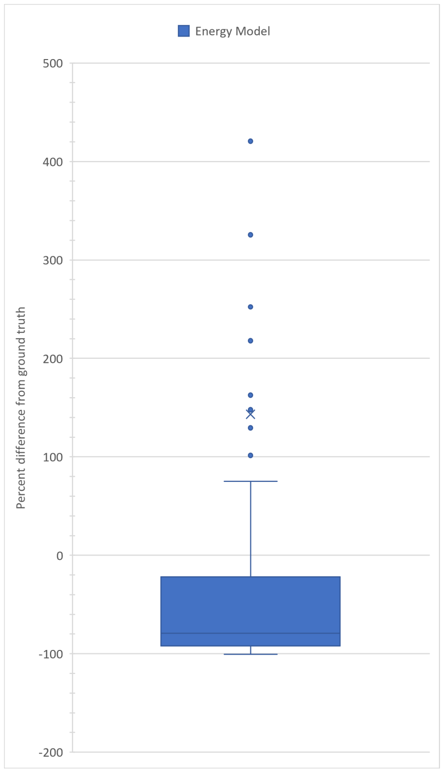
4.4.2 Time Consumption Results

When computing the energy consumption results, we also measure how much time it takes to compute each result. This means, for each benchmark we measure the time it takes for our interpreter to compute and count all CIL instructions, the time it takes to compute each machine learning prediction, and the time it takes to compute the energy consumption using the energy model. Furthermore, as all benchmarks are run using the `measurement` framework we also know how much time it takes to obtain a statistically significant measure of the run time.

An excerpt of the time results is first presented on Table 4.2.2 on page 64 and again below on Table 4.4.2 on page 69. The excerpt is representative for all time consumption results in that obtaining all machine learning results, except for random forest, takes less than a millisecond. For all benchmarks obtaining



(a) Lasso regression



(b) Energy model

Figure 4.4.2: Boxplot of Lasso regression and the Energy model

machine learning results with random forest takes 4-5 milliseconds. Furthermore, computing the energy consumption using the energy model also takes less than a millisecond for all benchmarks. To compare the time consumption of obtaining energy estimates statically to obtaining them dynamically, we recall that for all static techniques we must first compute the count of each CIL instruction in the program. The time for this is seen in the `Counts` column on Table 4.4.2. As the time taken to compute the energy consumption estimates using the machine learning models and the energy model is insignificant to the time it it takes to compute the count of each CIL instruction using our interpreter, we just compare this value to the dynamic. For our comparison we compute how many times faster it is to obtain the CIL counts than it is to obtain the statistically significant RAPL measurement, and we denote this value the *speedup*. For the programs in our benchmark suite the greatest speedup we observed was a 1280 times difference for the `csharp-program-concatenate-strings_0` benchmark and the lowest was a 5 times difference for the `Feigenbaum_constant_calculation_0`.

Table 4.4.2: Excerpt of the final time consumption results. All values are presented in ms and rounded to two decimals. For brevity the column names are shortened, such that `RF` means `Random Forest`, and `EM` means `Energy Model`

Name	***Counts***	***Linear***	***Lasso***	***Ridge***	***RF***	***SVR***	***EM***	***Dynamic***
example-of-a-...	272.00	0.12	0.13	0.20	4.87	0.09	0.05	78536.80
find-sum-of-all-...	287.54	0.12	0.17	0.37	5.00	0.90	0.60	36117.18
print-the-integer-...	410.30	0.12	0.12	0.25	5.00	0.09	0.07	35710.60
csharp-program-...	244.73	0,12	0.13	0.10	5.00	0.10	0.04	3640.00
Combinations_2	322.13	0.12	0.13	0.15	4.84	0.09	0.05	44901.60

Chapter 5

Reflections

This section provides a discussion of this entire project. First, we discuss the interpreter we create for .NET programs to count the CIL instructions being executed. Then we discuss the results obtained by estimating the energy consumption of a set of benchmark programs using both the machine learning approach and the energy model approach.

5.0.1 CIL Instruction Counting

For this project we create our own interpreter for CIL code to count the number of times specific CIL instructions are executed for a given path through a program. In the following section we discuss the implementation of our interpreter.

Prior to creating this interpreter, we discovered that open source interpreters to C# already exist. Our research has found MoonWalker[1] and the official .NET interpreter[2]. MoonWalker is developed on the Mono platform and is created as an equivalent to Java Pathfinder[3] for .NET programs. MoonWalker's last update is in 2009 and as such, due to version incompatibility we did not succeed in getting results from the program. Additionally MoonWalker only works on programs written using the Mono runtime[4] which is a different runtime to the official .NET runtime. The other interpreter we found was the official

[1]https://fmt.ewi.utwente.nl/tools/moonwalker/
[2]https://github.com/dotnet/runtime/blob/main/src/coreclr/vm/interpreter.cpp
[3]https://github.com/javapathfinder
[4]https://www.mono-project.com/

.NET runtime interpreter, which is still being maintained and updated. The interpreter is able to provide a summary of the executed CIL instructions for a program, which is what is desired for this project. The interpreter is written in C++ and resides in one file of 8000 lines. We have tried to utilise it, however given that no documentation could be found regarding the interpreter, we could not get it to run. Additionally, we tried to contact one of the maintainers of the .NET runtime GitHub repository for help on how to build and use the interpreter. They have not written back to us.

Limitations Of Our Interpreter

For this project, we have chosen to create an interpreter for CIL code in order to determine the amount of times specific CIL instructions are invoked for a given program. There are however features which are not supported in the interpreter, due partly to time constraints. However the lack of some features is also due to diminishing returns to implement them. This is the case for `multidimensional arrays`. The multidimensional arrays require a special form of constructor and separate methods to accessing elements and storing elements. There are however, not a lot of benchmarks which utilise this feature, and as such, it was decided not to support multidimensional arrays.

Another feature the interpreter lacks support for is `multi-threading`. This decision is mainly due to time constraints, as most of the benchmarks do not utilise the feature. `Exception handling` is also not supported, though it is mostly due to time constraints, as most of the groundwork for supporting them is laid. It can for instance, determine all of the exception handling blocks in the CIL code. To support this feature would require the addition of a return flag for methods called *Exception* as well as the handling of this return flag.

Aside from missing features, there are also some features which work to some extent, mostly in simpler cases, but will break down in others. In this case, the handling of `Generics` works with most small examples, however larger examples, found in the `LINQ` library, might not work as intended. Most input and output is also very lightly supported. Output is mostly set to perform no actions other than consume the arguments on the stack, as they rarely or never have any impact on the rest of the program.

Aside from the mentioned limitations in the implementation of features, there is another limitation to the accuracy of the resulting instruction count. It comes in the form of `Intrinsics`[5], which is an attribute that can be applied to methods, telling the compiler, that the specific method will be filled by the JIT compiler when executing. Since we have created our own interpreter, the intrinsic methods have to be handled. To simplify the implementation and the execution, we simply execute a replacement method, which does what the original method should do. However, using this approach, we loose some accuracy in the counting, because the replacement methods are not created using CIL instructions, but written directly into the interpreter.

5.0.2 Static Estimation Technique

For the static estimation techniques we have made several choices that can impact the overall results. First, we have chosen to use machine learning and create an energy model as means of static estimation. These two approaches are chosen as the literature showed several papers where similar approaches were implemented ([12–15]). However, it is interesting to also consider other approaches to estimating the energy consumption of programs. One such approach could be to estimate the energy consumption based on some time consumption model of programs. The literature shows ([3, 34]), that a slight correlation between time and energy exists, and this correlation could be interesting to model.

Another choice made for both static estimation techniques is, that they are based solely on the CIL instruction counts. This could however be extended. In [13, 14, 35], the authors estimate the energy consumption of a program by factoring three components. The base cost of the instruction, the cost of the transitions between two instructions and inter-instructions effects such as cache misses or pipeline stalls. For our results, only the base cost of each instruction is measured and used for estimation. Using the other two factors as in [13, 14, 35] could provide more accurate results. More specifically factoring transitions and inter-instructions effects would aid the underestimation seen in the results of the energy model and some machine learning models. For this project, we do not include the transitions between instructions and inter-instruction effects due to time limitations. To measure the transitions between instructions, this would require to measure two instructions in a row, and subtracting the cost for the instructions themselves. In addition, the current model does not account for exceptions thrown. Therefore, the behaviour of exceptions are indefinable

[5]`https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Runtime/CompilerServices/IntrinsicAttribute.cs`

in the interpreter and thus for benchmark that throws exceptions the estimates will be less accurate.

Lastly, common for both static estimation techniques is that the models they depend on are only computed on one specific CPU rendering them tailored to that CPU. However, as different CPUs may vary in energy usage per instruction the results only reflect the energy consumption of the CPU on which the energy model and the data set for the machine learning models are created. For more accurate results energy and machine learning models should be created for all CPUs.

5.0.3 Machine Learning Results

In Section 4.4 on page 64 the energy consumption values using the machine learning approach are presented. The results show, that the linear models, being linear regression, lasso regression, and ridge regression, perform the worst with the largest percentage difference from the ground truth. The non-linear models of random forest and support vector regression performs better than the linear. For random forest the minimum value is approximately -7.5% and the maximum value is approximately 9.2%. For the support-vector regression boxplot, the minimum and maximum are approximately -28% and 17% respectively. This is also seen in the boxplots shown on Figure 4.4.1 on page 66 and Figure 4.4.2a on page 68. Furthermore, the linear models are able to predict negative values for the energy consumption. This is naturally non-desirable as a negative energy consumption is unrealistic. This suggests that the data is not linear or that not enough data has been collected to fit the linear model properly. These results confirm the hypothesis presented in Section 4.1 on page 55

Several measures can be implemented to improve the machine learning models. First, the data set consists of 147 programs, which is less than the total number of features, being the 226 different CIL instructions. When the number of observations is less than the number of explanatory variables, there can be a lot of variability in the least squares fit (see Section 2.2.4 on page 13), resulting in overfitting and consequently poor predictions on observations not used in training the model [36]. Therefore, to improve on the machine learning model a larger data set is needed. A larger data set could be obtained by improving the interpreter such that it can handle more benchmarks. Furthermore, to improve the machine learning models it could be interesting to inspect the level of correlation between each feature and the energy consumption. This could reveal which features are highly correlated and which are not. The less correlated features can then be discarded resulting in regression models with fewer but more correlated features. In [37] the author states that a *good feature sets contain features that are highly correlated with the class, yet uncorrelated with each other*, which is what we would strive to achieve. Furthermore, new features could be introduced that help predict the energy consumption of the program. For example in [12] they use performance features such as cache hits and misses and branch prediction successes. This however requires the program to be run, which defeats the purpose of the static estimation. Instead data such as the transition between instructions could be included into the data set.

Furthermore, besides the fact that we have excluded programs in the data set that contains commandline arguments and runtime IO, some programs still contain file IO. This type of IO cannot be handled by the interpreter which merely disregards the instructions for the IO. In this way, some programs are included in the data set are not correctly represented. This is because they are represented with the correct energy consumption as measured by our measuring framework but, the instruction counts are incorrect due to our interpreter disregarding file IO. Because of this some entries in the data set can be considered faulty.

5.0.4 Energy Model Results

The results from the energy model are seen in Section 4.4 on page 64, where it is seen that the median is -79.22%. This means, that in general the energy model overestimates the energy consumption of the programs, which contradicts our hypothesis made in Section 4.1 on page 55. Furthermore, based on Q1 and Q3, being -91.9% and -21.98% respectively, the average results from the energy model do not convey much information about the actual energy consumption of the program analysed.

To investigate why the performance of the energy model is worse than anticipated, we consider the instructions and the composition of the energy model. In terms of composition we create the energy model as a linear combination of how many times each instruction is encountered in a program by our interpreter and the instruction's energy cost. As seen in Section 1.2 on page 2 and discussed in Section 5.0.2 on page 72 it is commonplace in the literature to include the transitions between instructions and other energy cost such as cache misses and pipeline stalls.

Considering the instructions, not all 226 instructions are implemented and not all implemented instructions can be estimated precisely. Firstly, a list of the CIL instructions not implemented is seen in Appendix A on page A-1. The list includes in total 50 instructions of 226. However only 14 of these instructions are present in the programs we use for benchmarking. The instructions not implemented include instructions that depend on some address in memory, such as all `ldind` instructions. Furthermore, instructions related to exception handling are not implemented as the instruction `Throw` cannot be emitted without also being caught before stopping the measurements. Thus, we cannot get an accurate measurement of either. As these instructions are not implemented this results in the estimations from the energy model underestimating the ground truth. However, as only 14 of the 50 not implemented instructions are actually encountered in the benchmark programs we do not entirely attribute the underestimation of the energy model to this. Lastly, from Appendix A on page A-1 we see that the `pop` and `ret` instructions are not implemented per se. However, they are indirectly measured, as when measuring all instructions, the `DynamicMethod` to which we emit the instructions must return, thus all emit a `ret`. Also, when loading a value this value must eventually be popped from the stack before returning. Therefore, when measuring the energy consumption of the `load` we also measure the energy of the `pop` instruction.

Considering the instructions that *are* implemented some instructions are difficult to estimate precisely. This includes the data dependent instructions such as `mul`. In the case of the `mul` instruction the energy consumption of the instruction depend on the input given [13]. To measure the energy consumption of instructions of this kind we use the framework that we created in Section 3.3.1 on page 47 that builds on the `measurement` framework of [3]. Using this framework allows us to provide randomised values to such instructions. However, for a more precise measurement, the method for measuring the energy consumption of data dependent instructions could be split into several separate measurements accounting for values of different sizes. In this way, we could for example measure the energy consumption of the `mul` instruction for large integer values and for small integer values. If our interpreter could provide a range for the values for each data dependent instruction, the results could be more precise as the more specialised instruction measurement could be used. Another option is to create a custom attribute which the user could provide to each integer with the expected range.

Furthermore, the created data set contains negative values for the energy consumption of some instructions. This is because, to measure the energy consumption of some instructions, they require some stack transitional behaviour before being able to execute. Thus, we need other instructions to setup the stack before executing the desired instruction, the energy consumption of the additional instructions need to be subtracted from the result of the instruction that we are measuring. If the combined additional results gets larger than the result for this method, then it leads to the result being a negative number. The combined additional result should not be able to grow larger then the result. However, as the benchmarks for the instructions are small, and take less than 0.1 ms to run, the results can be influenced with the operating system.

Lastly, as mentioned in Section 2.2 on page 8, another approach to computing the energy model is to create a series of C# programs instead of using `System.Reflection.Emit` to generate CIL code to test the CIL-instruction set. This would ease the process of creating the different programs needed, as the stack would not need to be prepared before each instruction that requires it.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This project documents how we created an extension for Visual Studio Code that aids developers in reasoning about the energy consumption of their software. We created three approaches for estimating the energy consumption of programs, being statically using machine learning, statically using an energy model, and dynamically using RAPL. Furthermore, to aid the static approaches we created an interpreter for CIL code. However, as the results showed, neither of the static approaches, besides random forest regression, were able to accurately estimate the energy consumption, though we present suggestions for improvements in Chapter 5 on page 70. Both static and the dynamic estimation approach were implemented in the Visual Studio Code extension in the form of a micro service architecture. We created a service for the machine learning estimate, one for the energy model and one for the dynamic approach.

This project was created and guided by the problem statement in Section 1.1 on page 2 which is reiterated below. The following sections summarise the entire project and answer the problem statement.

How can an IDE extension be made to aid developers in reasoning about the energy consumption of their software

First, we explored the related work of this field in Section 1.2 on page 2, to investigate the current state of research. Several tools were found which aim to estimate the energy consumption of programs. Some of which use program analysis at runtime and some perform the analysis statically. Another tool uses per-instruction energy modelling and incorporates the estimation into an extension for the IDE Eclipse. In addition to tools, we also found the current state of estimation techniques. These techniques include machine learning based on software performance features such as cache misses and context switches. Other papers created per-instruction energy modelling for Java, which measured the energy consumption of performing each Java Opcode. These measurements were used along with an interpreter to estimate the energy consumption of programs or to create a probabilistic energy distribution of instructions. Lastly, the related work also showed dynamic approaches using RAPL where a program is run multiple times while the energy is measured.

In Chapter 2 on page 5 we determine the specifics of the project and how we answer the problem statement. We designed a user interface for the Visual Studio Code IDE extension. A prototype of the extension was created, being a Visual Studio Code WebView that conformed to the general design of Visual Studio Code. From the user interface we allow the user to choose for which programs or methods they want energy consumption estimates. They can also choose how they want to obtain the measurements, be it statically using the energy model or machine learning or dynamically using RAPL.

We also designed how to implement the approach for obtaining the energy consumption estimates. For the energy model and the machine learning we explored different software abstractions to use as a basis for the estimations, being C# source code, CIL code, and machine code. We decided to use CIL code as this provided a restricted set of instructions while being machine independent. Then, to count CIL instructions, we decided to create an interpreter for CIL code to count all CIL instructions in a program along with the number of times each CIL instruction is performed. For the machine learning models we investigated five different regression techniques to predict the energy consumption of programs. We decided to employ an experimental approach to determine which of these techniques performs best on the problem of this project. We decided to create the energy model using run time code generation in C#. Lastly, for the dynamic approach we decided to extend the framework created in our previous work [3].

Following the design, we implemented the different estimation techniques and the Visual Studio IDE extension in Chapter 3 on page 27. We implemented the IDE extension according to our designed prototype. For the extension to get the energy estimates we created a micro-service architecture where microservices are created for each estimation approach. We also created a microservice for our interpreter. In this way, the extension can communicate with the microservices using HTTP calls as needed, and the microservices for the estimation approaches can communicate with the interpreter microservice as needed.

Our CIL interpreter was created using Python with the purpose of providing a count for each executed instruction. Along with the interpreter, the machine learning models were created using Python, and trained on a set of 147 programs. For the per-instruction energy modelling approach, we used run-time code generation in C#, using the `System.Reflection.Emit` library, to emit individual instructions and thus measure their energy consumption. As instructions must be emitted to a method, we used the approach of creating `DynamicMethods`, which are also part of the `System.Reflection.Emit` library. We created `DynamicMethods` for each CIL instruction. Then the energy consumption of the methods were measured, and any dependencies subtracted. The measurements were then used for the implementation of the energy model estimation. Lastly, the dynamic estimation approach was created based on the measurement framework made in our previous work [3].

Based on the implemented estimation approaches, we determined the error of each static approach in Chapter 4 on page 55. We used the dynamic estimates as the ground truth as this approach measured the energy consumption of each benchmark program and provided statistically significant results. We calculated the error of a benchmark as the percentage deviation from the ground truth. Based on these results, the non-linear machine learning estimates had a lower percentage deviation from the ground truth than the energy model and all of the linear machine learning models. Furthermore, the energy model had a lower percentage deviation from the ground truth than the linear models, except for lasso regression. The estimation approach with the least error was the random forest machine learning model, with a minimum of -7.49% and a maximum of 9.19%. The median of random forest is 1.06%, which indicates a slight overestimation of the energy consumption.

6.2 Future work

This section provides an overview of future work which can be made, based on this project, to encourage future research in this field. Here, multiple possible areas of future work is elaborated upon and described.

6.2.1 Improved interpretation approach

In Section 2.2 on page 8, two different estimation approaches are designed. Being the *naive approach* and the *Counting by Interpretation* approach. The interpretation is the approach used throughout the results. An alternative to the interpretation approach could be to use more tools from static analysis such as symbolic execution. One idea for such an approach is to use a single pass of the CIL program to create a control flow graph (CFG) and then, based on the graph, generate the set of input values necessary to explore all code paths. This would make it a more generic energy estimation approach compared to the interpreter, as the interpreter only follows a single path through the program, either through input parameters or hard-coded values. Taking the average of all paths could provide a better estimate, than simply executing one path. The advantage of such an approach is the speed obtained by not interpreting every single instruction. One disadvantage of this approach is the large set of possible paths even for small programs, especially when loops are involved.

6.2.2 Using the .NET interpreter

As discussed in Chapter 5 on page 70, the .NET repository for the official runtime contains its own interpreter. This interpreter is created to run C# programs, with the added feature of summarising the number of times each CIL instruction is executed. Compared to the interpreter we have created, we anticipate that the official .NET interpreter would provide more accurate counts of the CIL instructions, since it is kept up to date with the rest of the runtime code and it supports all CIL instructions. Thus, if we could get the official interpreter to work, then more benchmarks would be available for training and testing.

6.2.3 Improve Machine Learning Models

For future work it would be relevant to improve the machine learning models. Ideas for how this could be done are presented in Chapter 5 on page 70 and elaborated on here. Firstly, we propose that a larger data set could contribute to better results. In total 1438 programs are scraped from the internet to be used for the data set, however only 147 programs are part of the final data set. This is because we exclude programs based on certain criteria as presented in Section 3.2.2 on page 34. Thus to expand the data set more programs must comply to the criteria. In Section 3.2.2 on page 34 we also present a pie-chart of the criteria and the percentages with which they exclude programs. The criteria that make us exclude the most programs are those stating that the programs must include a `Main` function (22.3 %) and that the program must be able to be run with our interpreter (35.9 %). For the programs that do not have a `Main` function we could write a script to automatically refactor those programs such that the program is started from a `Main` function. For the criterion that states that the program must be able to be run by our interpreter we exclude 35.9 % of all programs. To combat this, we must improve the interpreter or use the .NET interpreter instead. Suggestions for how to do this are described above.

Another suggestion for improvement of the machine learning models is to limit the number of explanatory variables, which is currently 226. This would make the data set more concise, but possible also more precise. To do this, we would investigate which variables are mostly correlated to the output variable.

6.2.4 Improving the Energy Model

In Chapter 4 on page 55 the values from the energy model are shown as a boxplot on Figure 4.4.2b on page 68. From the graph we see that the model underestimates in most cases. However, the energy model also only takes into account how many of each instruction is present multiplied by the energy consumption of the given instruction. This is a simple model with a linear mapping between instructions and total energy consumption. In the literature an energy model is often presented as the product of the energy consumption of each instruction and the number of times they are executed, the energy consumption of the transition between instructions, and inter-instruction effects. We only model the first part of this. Thus, for future work we would implement the remaining parts of the product into our energy model. To model the transition between instructions, we would count the number of times each instruction switches to any other instruction and then multiply that number with the energy consumption of the transition. This should be done for all transition pairs of instructions. Lastly, we would model the inter-instruction energy costs associated with the program, being for example the effects of cache misses. Especially, we anticipate that modelling the transitions will improve the estimation.

Furthermore, as stated in Chapter 5 on page 70, not all CIL instructions have been measured. Therefore, to improve the energy model, it would be ideal to implement these instructions. Since some of the missing instructions are encountered in the benchmarks, they are part of the reason the energy model underestimates.

6.2.5 Usability Test of IDE Extension

For future work we find it relevant to conduct usability tests of our IDE extension. This would provide insights into how it could be improved such that developers benefit the most from the extension. The usability test should be conducted with subjects that are within the target group of the extension, that is developers that create software where the energy consumption is of concern [2].

6.2.6 Creating An Energy Model

The energy model estimation technique uses an XML file describing the energy consumption of each instruction in CIL. However, different CPUs energy consumption for instructions can vary. Therefore, the energy consumption of a program can vary based on, on which CPU the program is run, and the energy model should reflect this to achieve the best estimates. As such, for the best estimation results the energy model XML file should be created on the same system, for which the estimates are performed. For future work, it should be readily available for the user to create an energy model to reflect their system, such that the user can get the most accurate results.

Bibliography

[1] Eric Masanet et al. "Recalibrating global data center energy-use estimates". In: *Science* 367.6481 (2020), pp. 984–986. ISSN: 0036-8075. DOI: 10.1126/science.aba3758. eprint: https://science.sciencemag.org/content/367/6481/984.full.pdf. URL: https://science.sciencemag.org/content/367/6481/984.

[2] Irene Manotas et al. "An Empirical Study of Practitioners' Perspectives on Green Software Engineering". In: *Proceedings of the 38th International Conference on Software Engineering.* ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 237–248. ISBN: 9781450339001. DOI: 10.1145/2884781.2884810. URL: https://doi.org/10.1145/2884781.2884810.

[3] Anne Ejsing et al. "The Influence of Programming Paradigms on Energy Consumption". MA thesis. Aalborg University, Jan. 2021, p. 109.

[4] Ward Van Heddeghem et al. "Trends in worldwide ICT electricity consumption from 2007 to 2012". In: *Computer Communications* 50 (2014). Green Networking, pp. 64–76. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2014.02.008. URL: https://www.sciencedirect.com/science/article/pii/S0140366414000619.

[5] Statista. *Internet of Things (IoT) and non-IoT active device connections worldwide from 2010 to 2025 (in billions) [Graph]*. 2020. URL: https://www-statista-com.zorac.aub.aau.dk/statistics/1101442/iot-number-of-connected-devices-worldwide/ (visited on 09/06/2021).

[6] Stijn Shuermans and Christina Voskoglou. *The Global Developer Population 2019 - How many developers are there?* 2019. URL: https://

slashdata-website-cms.s3.amazonaws.com/sample_reports/EiWEyM5bfZe1Kug_.pdf (visited on 09/06/2021).

[7] Candy Pang et al. "What do programmers know about the energy consumption of software?" In: (July 2015). DOI: 10.7287/PEERJ.PREPRINTS.886.

[8] G. Wei et al. "FPowerTool: A Function-Level Power Profiling Tool". In: *IEEE Access* 7 (2019), pp. 185710–185719. DOI: 10.1109/ACCESS.2019.2961507.

[9] S. Hao et al. "Estimating mobile application energy consumption using program analysis". In: *2013 35th International Conference on Software Engineering (ICSE).* 2013, pp. 92–101. DOI: 10.1109/ICSE.2013.6606555.

[10] M. Couto, J. Saraiva and J. P. Fernandes. "Energy Refactorings for Android in the Large and in the Wild". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER).* 2020, pp. 217–228. DOI: 10.1109/SANER48275.2020.9054858.

[11] Kashif Nizam Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (Mar. 2018). ISSN: 2376-3639. DOI: 10.1145/3177754. URL: https://doi.org/10.1145/3177754.

[12] C. Fu, D. Qian and Z. 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).* 2018, pp. 490–496. DOI: 10.1109/Cybermatics_2018.2018.00106.

[13] James Pallister et al. "Data Dependent Energy Modeling for Worst Case Energy Consumption Analysis". In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems* (June 2017). DOI: 10.1145/3078659.3078666. URL: http://dx.doi.org/10.1145/3078659.3078666.

[14] V. Tiwari et al. "Instruction level power analysis and optimization of software". In: *Proceedings of 9th International Conference on VLSI Design.* 1996, pp. 326–328. DOI: 10.1109/ICVD.1996.489624.

[15] Sébastien Lafond and Johan Lilius. "An Energy Consumption Model for an Embedded Java Virtual Machine". In: *Architecture of Computing Systems - ARCS 2006*. Ed. by Werner Grass, Bernhard Sick and Klaus Waldschmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 311–325. ISBN: 978-3-540-32766-0.

[16] Murali Annavaram. "Energy per instruction trends in. Intel microprocessors". In: *Technology Intel Magazine* (2006).

[17] Kashif Nizam Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (Mar. 2018). ISSN: 2376-3639. DOI: 10.1145/3177754. URL: https://doi.org/10.1145/3177754.

[18] Stefanos Georgiou and Diomidis Spinellis. "Energy-Delay investigation of Remote Inter-Process communication technologies". In: *Journal of Systems and Software* 162 (2020), p. 110506. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.110506. URL: https://www.sciencedirect.com/science/article/pii/S0164121219302808.

[19] S. Georgiou et al. "What are Your Programming Language's Energy-Delay Implications?" In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 2018, pp. 303–313.

[20] .NET Foundation. *RyuJit Overview*. URL: https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/jit/ryujit-overview.md (visited on 11/05/2021).

[21] Andriy Burkov. *The Hundred-Page Machine Learning Book*. Andriy Burkov, 2019. ISBN: 978-1999579500.

[22] Yale. *Linear Regression*. 1997. URL: http://www.stat.yale.edu/Courses/1997-98/101/linreg.htm (visited on 21/03/2021).

[23] Matti Pirinen. *RIDGE REGRESSION VS ORDINARY LEAST SQUARES (OLS)*. 2020. URL: https://www.mv.helsinki.fi/home/mjxpirin/HDS_course/material/HDS6_slides.pdf (visited on 21/03/2021).

[24] Alex Smola and Bernhard Schölkopf. "A tutorial on support vector regression". In: *Statistics and Computing* 14 (Aug. 2004), pp. 199–222. DOI: 10.1023/B\%3ASTCO.0000035301.49549.88.

[25] Peter Sestoft. "Runtime Code Generation with JVM and CLR". 2002. URL: http://www.itu.dk/people/sestoft/rtcg/rtcg.pdf.

[26] Microsoft. *IL Generator Class*. 2021. URL: `https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.ilgenerator?view=net-5.0` (visited on 16/03/2021).

[27] Microsoft. *DynamicMethod Class*. 2021. URL: `https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.dynamicmethod?view=net-5.0` (visited on 16/03/2021).

[28] Vivek Tiwari et al. "Instruction level power analysis and optimization of software". eng. In: *Journal of VLSI signal processing* 13.2-3 (1996), pp. 223–238. ISSN: 0922-5773.

[29] Microsoft. *Extension API*. 2021. URL: `https://code.visualstudio.com/api` (visited on 04/03/2021).

[30] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. 6th ed. Geneva, Switzerland, June 2012. URL: `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[31] AIOHTTP. *Asynchronous HTTP Client/Server for asyncio and Python*. URL: `https://docs.aiohttp.org/en/stable/` (visited on 15/04/2021).

[32] Microsoft. *Writing Custom Attributes*. 2018. URL: `https://docs.microsoft.com/en-us/dotnet/standard/attributes/writing-custom-attributes` (visited on 30/03/2021).

[33] William G. Cochran. *Sampling Techniques, 3rd Edition*. John Wiley, 1977. ISBN: 0-471-16240-X.

[34] Rui Pereira et al. "Energy Efficiency across Programming Languages". In: (2017). DOI: `10.1145/3136014.3136031`. URL: `https://doi.org/10.1145/3136014.3136031`.

[35] Kerstin Eder and John Patrick Gallagher. "Energy-Aware Software Engineering". English. In: *ICT - Energy Concepts for Energy Efficiency and Sustainability*. Ed. by Giorgos Fagas et al. InTechOpen, Mar. 2017, pp. 103–127. ISBN: 978-953-51-3011-6. DOI: `10.5772/65985`.

[36] Gareth James et al. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. ISBN: 1461471370.

[37] Mark A. Hall. “Correlation-based Feature Selection for Machine Learning”. PhD thesis. The University of Waikato, Aug. 1999. URL: `https://www.cs.waikato.ac.nz/~ml/publications/1999/99MH-Thesis.pdf`.

A Missing CIL instructions for the energy model

- Arglist
- Calli
- Castclass
- Constrained
- Cpblk
- Cpobj
- Endfilter
- Endfinally
- Initblk
- Initobj
- Isinst
- Ldelema
- Ldfld
- Ldflda
- Ldftn
- Ldind_I
- Ldind_I1
- Ldind_I2
- Ldind_I4
- Ldind_I8
- Ldind_R4
- Ldind_R8
- Ldind_Ref
- Ldind_U1
- Ldind_U2
- Ldind_U4
- Ldloca
- Ldloca_S
- Ldobj
- Ldsfld
- Ldsflda
- Ldtoken
- Ldvirtftn
- Leave
- Leave_S
- Localloc
- Mkrefany
- Readonly
- Refanytype
- Refanyval
- Rethrow
- Starg_S
- Stelem_Ref
- Stfld
- Stind_Ref
- Stobj
- Switch
- Throw
- Unaligned
- Volatile