Benchmarking C# for Energy Consumption

Energy Implications of Different Language Constructs and Collections

Pre-specialization in Software

Project Report
CS-21-PT-9-02

Aalborg University
Software

Software
Aalborg University
https://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Benchmarking C# for Energy Consumption

Theme:
Energy Aware Programming

Project Period:
Fall Semester 2021

Project Group:
CS-21-PT-9-02

Participant(s):
Aleksander Øster Nielsen
Kasper Jepsen
Lasse Stig Emil Rasmussen
Milton Kristian Lindof
Rasmus Smit Lindholt
Søren Bech Christensen

Supervisor(s):
Bent Thomsen, Lone Leth Thomsen

Copies: 1

Page Numbers: 167

Date of Completion:
December 28, 2021

Abstract:

This project investigates the impact of different C# language constructs on energy consumption. To accomplish this, we create a framework that is capable of measuring the energy consumption of benchmarks and determining if there is a significant difference between benchmarks in the same group. The framework uses Intel's RAPL to collect the energy consumption of the benchmarks. We use the framework on a suite of 316 benchmarks across 16 groups divided into 56 subgroups, to generate results for how the different language constructs compare to one another in regards to both energy consumption and run-time. These results can be used by developers to choose what language constructs to use in different situations. Given the results, we find an explanation for differing language construct behavior. This is accomplished in a four-step process which is stopped when an explanation is found. To start the process we examine differences in the benchmarks' code, then the documentation for constructs, differences in the Intermediate Language (IL) code, and lastly differences in the assembly code.

Preface

This project has been created by 6 software students in the ninth semester, at Aalborg University, under the theme *Energy Aware Programming* in the period 2nd of September 2021 to 3rd of January 2022. We want to thank our supervisors Bent Thomsen and Lone Leth Thomsen for their guidance throughout this project.

Contents

Chapter 1

Introduction

Energy consumption from Information and Communications Technology (ICT) systems has increased rapidly in recent years. The increasing energy demand from this sector has increased the environmental and societal concerns. In [1] it was reported how energy consumption from data centers totaled 205 TWh in 2018, which was 1% of global electricity consumption and an increase of 6% compared to 2010. This amount is expected to rise in the coming years [2].

Some reduction of energy consumption of ICT can be accomplished by writing software in an Energy-Aware fashion. Research in regards to energy consumption in software has shown that many choices impact energy efficiency, ranging from choices of datatypes to algorithm choice and coding style [3, 4, 5, 6].

To allow developers to make energy-efficient applications they need to have knowledge and tools to make energy-conscious choices. There exists research into several different subjects that help achieve better energy efficiency. Some look into the energy efficiency of different languages used for the same programming problems, others research the effect of different programming paradigms in multi-paradigm languages and the differences between interpreted and compiled languages [7, 8, 9].

Even with this research into Energy-Aware Programming, the literature is sparse when it comes to research into other parts of languages, such as the specific language constructs as well as libraries made for these languages. The lack of a body of knowledge poses a challenge for determining the impact of language constructs [10]. The main research into language constructs focuses on the programming language Java, meaning that there is a lot of research to be done outside this language [4, 11].

As previous research is predominantly focused on Java, we seek to expand the field of research by looking at another language. This leads us to look at other commonly used languages. According to [12], C# is the second most used Object-Oriented Programming (OOP) language as of December 2021. Given that it is such a prevalent language we choose to conduct research within it, as results found conducting this research can provide greater impact than choosing a less prevalent language. Thereby, we investigate how different language constructs affect the energy efficiency of C# programs. We generate energy measurements for different language constructs in C#, which can facilitate better energy utilization. This investigation provides a further basis for developing guidelines and tools that help facilitate energy awareness when programming in the C# language and provides groundwork for future research into C# regarding energy awareness.

1.1 Problem Statement

This section presents the problem statement for the project and the motivation behind it. Given the information discussed in the introduction, we create the following problem statement and the accompanying research questions:

What are the energy consumption effects of different language constructs in C#?

1. *How can the energy efficiency of language constructs be measured?*
2. *How can a suite of experiments/benchmarks be made to measure the energy efficiency of language constructs?*
3. *Why is the energy consumption different between similar language constructs?*

The problem statement is motivated by the lack of research into C# energy consumption and the multitude of different language constructs available within the C# language. The research conducted in [4] and [11] shows that different language constructs affect the energy efficiency of programs. The research conducted in these two papers is done in Java and as such we extend that work and look at the effect of language constructs in C#.

The first research question makes certain that the method we use to obtain results is reliable, meaning the results can be replicated. The second research question ensures that the experiments are constructed in a manner that spans a variety of language constructs.Research question three gives understanding of why similar constructs have different energy consumption. By using this understanding we may then bring awareness to developers explaining which constructs are preferable in regards to energy efficiency.

We examine related works within the Energy-Aware programming field to get a proper understanding of the state of the art in Chapter 2. The chapter builds a picture of the current state of the field and informs us of what is missing within the field. Afterward, we explore language constructs to understand which language constructs are relevant in Chapter 3. In Chapter 4 we divulge our thoughts for how we design the experiments and why we design them this way. In Chapter 5 we look at how we implement and execute the experiments. Following we have Chapter 6, where we show the interesting or unexpected results of the experiments. Then in Chapter 7 we analyse the results from Chapter 6 to further understand the behaviour of the language constructs. The analysis is followed by our reflections from the duration of the project in Chapter 8. The reflections are followed by the conclusion of the project in Chapter 9, where we explain the contributions of the project. At the end in Chapter 10, we explain future possibilities that can complement the conclusions of this project.

1.2 Work Process

As we have specified our problem statement, we now discuss our work process.

During the approximately four-month duration of our project, our work process is a hybrid of the plan-driven and agile processes [13]. We classify it as such because we create a plan, follow that plan and adapt as needed from unforeseen challenges or inspirations. We do not make large iterations over our product, which is the framework for measuring benchmarks, as well as the benchmarks since the limited time does not permit this. The report is iterated over multiple times as we get feedback from supervisors and project members reading the content. We perform daily stand-up meetings to keep up with what is currently going on in the project and what needs to be done in the near future. We do it in part to have an understanding of

what we expect to get done in the coming days. In addition, we discuss if someone is stuck with any problems, which we then as a group can discuss or assign someone to help solve the problem.

We do not plan the whole project from the start, as a lot of variables change during a semester, such as the precise direction of the project based on the research we make, which language constructs to test, etc. In this way, we use something similar to agile, where we plan the near future and have ideas of what is in the future. Major milestones such as "report needs to be in a deliverable state" exist at specific dates, to make sure we meet the hard deadline associated with the project.

To keep track of what each other is doing, we use a kanban board, specifically, we use `GitKraken Boards` [14]. The board contains cards with required tasks, and we can assign ourselves to the cards, and move them between different categories like to-do, in progress, and done.

Besides this, we use `GitLab` to keep track of our code repository, so we can share and keep track of changes to the code that is produced throughout the project.

Chapter 2

Background & Related Work

In this chapter, we investigate different means of power measuring for computer systems and get an understanding of the related work. We then look at the research regarding the energy consumption of language constructs. Additionally, we look at tools seeking to ease the process of programming with energy efficiency as the focus.

2.1 Power Measuring

To investigate the power consumption of a computer system, there are two main options [15, 16]. One option is the use of an external hardware-based measuring approach. This is usually done with a power meter plugged into a wall-socket. As the second option, there is software-based measuring. In this approach, the power measuring happens on the processor of the system that is being measured. As part of looking at these different options, we look at the granularity of their measurement methods. The information is then utilized during experimental design in Chapter 4.

2.1.1 Hardware-based Power Measuring

The power consumption of a computer system can be measured through the use of external hardware tools. These tools measure power at the wall-socket, meaning they measure the power of the entire system. We investigate two different wall-socket measurement tools, the *Watts Up Pro* and *Kill A Watt*. We look at *Kill A Watt* as an alternative to *Watts Up Pro* as it has been discontinued, and *Kill A Watt* is suggested as an alternative [17].

Additionally, we investigate the tool PTDaemon created by SPEC that can be used together with wall-socket measurement devices.

Watts Up Pro is a hardware-based solution for power measuring [18]. This power meter can monitor real-time electricity usage and can be used with logging software to store the logs on a computer. This is done by plugging the Watts Up Pro into a wall-socket and then plugging the device that is to be measured into the Watts Up Pro. The Watts Up Pro is then plugged into a computer using a USB cable to use the logging software. It can monitor four different energy units without needing to convert from one unit to another, these being Ampere, volt, watt, and kilowatt. Depending on the units being logged, Watts Up Pro can store from 1000 up to 32000 records [18]. The minimum logging interval is once per second and has an accuracy of $\pm$ 1,5% [18]. The nature of the measuring device means that the entire system is being measured.

Kill A Watt is a wall socket electricity usage monitor which measures the energy usage of a device that is plugged into the meter. The meter has a screen on which the different readings of the meter can be read. The meter can measure voltage, current, energy consumed in kWh, and hours connected. Some models display estimated costs. The meter updates its display every second and has an accuracy of $\pm$ 0,5% [19]. The meter cannot store, transmit or transfer its readings. As such Kill A Watt has limited usage for any ongoing monitoring purposes. Hobbyists have improved upon Kill A Watt allowing it to connect to a Raspberry Pi, which allows for storing of the measured data [19, 20].

PTDaemon is a tool based on TCP-IP protocol. PTDaemon is created by Standard Performance Evaluation Corporation (SPEC). The interface can be used to offload power measuring or temperature sensor control from a system under test to a controller system. PTDaemon can connect to multiple power measuring devices and temperature sensors. SPEC has created a list of supported devices [21]. The list is periodically updated by PTDaemon to support new power measuring devices. For a Power measuring device to be accepted, it goes through an acceptance process created by SPEC [22]. As part of the process, the device specifications are checked, software support is added and different tests are performed. PTDaemon is used in multiple SPEC benchmarks. This includes SPECpower_ssj 2008, SPEC SERT suite, and Chauffeur WDK [23].

2.1.2 Software-based Power Measuring

A multitude of Power Measuring Application Programming Interface (API)'s exist for different platforms and system architectures. We choose to only consider API's available for x86/x86-64 systems, as most servers in data centers and personal computers use this architecture [24, 25]. In the mainstream desktop and laptop market, there are two significant CPU providers for x86/x86-64 systems, namely Intel and AMD [25]. Because of this, we look into Intel's Running Average Power Limit (RAPL) and AMD's Application Power Management (APM).

RAPL is an interface created by Intel and can be used with their CPUs [26]. Intel introduced it with their Sandy Bridge line of processors [26]. RAPL provides the ability to set custom power limits for the processors' package. It provides a way to access energy consumption information which can be used when measuring the power consumption of running code. RAPL has been used and tested in different studies. In earlier versions of RAPL and thereby on earlier architectures, the accuracy and stability were not great [27], but later revisions have been improved and are shown to be both accurate and stable [28, 29]. In [27] they conclude that with the Haswell architecture, the readings from RAPL were comparable to ones found from wall-socket power measurement. A later microarchitecture generation Skylake is an improvement upon Haswell, with added accuracy and additional monitored domains [27]. There is a limitation regarding the operating systems that can be utilized in conjunction with RAPL, as it can only be interfaced with on Linux systems [27]. RAPL per default updates the energy consumption once every 1 ms and accumulates the power consumption.

RAPL defines a set of Power domains [26]. Each domain is a named collection of components whose power usage can be measured. The domains found in most Intel CPUs are *Package* (PKG), *Power Plane 0* (PP0), *Power Plane 1* (PP1) and *DRAM*.

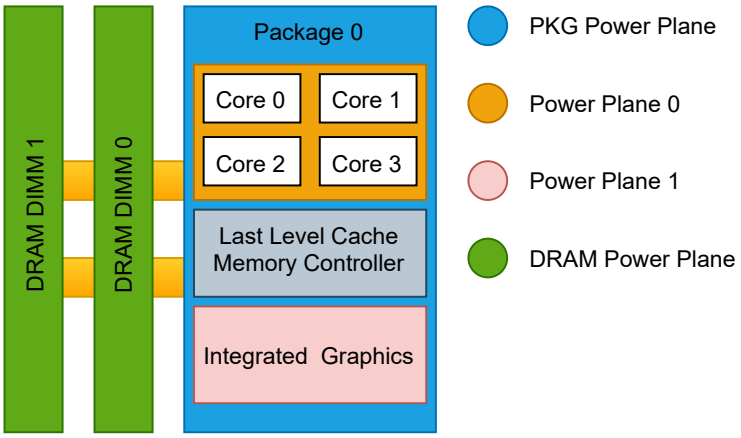

Figure 2.1: RAPL domains.

The different domains are illustrated in Figure 2.1. The domain PKG is used to measure the energy consumption of the entire CPU package. The domain PP0 measures the CPU cores exclusively, and PP1 is used to measure the uncore devices. Uncore devices are components that are not inside the cores, for example, the integrated graphics on a CPU. The last domain, DRAM, measures the dynamic random-access memory power consumption. This means that $PP0 + PP1 \leqslant PKG$. DRAM is independent of the other domains [26].

AMD's APM allows for an AMD processor to restrict its power usage, formally called TDP limiting as well as calculating the available power for turboing of the processor. It was introduced with the 15h family of processors [30]. Furthermore, it can dynamically monitor core activity and on that basis generate a power consumption approximate [30]. APM per default updates the energy consumption every 10 ms. These 10 ms is called a frame, and a special register on the CPU contains the average power consumption over that frame. Here the main difference between AMD APM and RAPL can be seen, as RAPL updates energy consumption, by default, every 1 ms. AMD has made a RAPL implementation on their newer Zen 2 architecture CPU's that can be accessed, however, there are shortcomings such as no DRAM domain and the implementation is shown to be inaccurate [31].

2.2 Related Work

The focus of this project is language constructs in C# and how knowledge about these can be used for optimizing software energy consumption. Our area is only a small part of Energy-Aware Programming as it spans several areas of research beyond this. To illuminate current research within Energy-Aware Programming, we provide an overview of a few of the areas that others have researched. In addition to this, we look at what research exists concerning energy consumption and language constructs, look at tools that help with energy optimization, and delve into two papers concerning language constructs and energy consumption in Java.

2.2.1 Research Overview

In this overview of current research, we look at several areas, however it is not an exhaustive list of all research. First, we look at research into metrics for Energy-Aware Programming. As the next area we look at how different hardware measuring methods compare. We follow this with a look at Software Engineering, this is both regarding utilized models and the knowledge that exist within the field regarding energy consumption. We then look at energy measurement both in regards to existing approaches and creating new alternatives. As the next part, we look at methodology and tools for energy consumption benchmarking. Next, we move on to research regarding how efficient the energy consumption of a language is and how execution-time may relate. Moving on we look at how different API's and collections influence energy consumption, following this we look at a paper that presents research into automatic optimization of energy consumption. As the last part of this overview, we look into how the maintainability of code may be influenced by changes that are concerned with energy consumption and vice versa.

In [32] there is research into metrics for Energy-Aware Programming, specifically it looks at presenting a uniform way of communicating measurements to facilitate stakeholder comprehension. To accomplish this the paper presents a metric that expresses a score for the resource utilization. This score is called *Resource Utilization Score* and allows for objective comparison of application configurations and versions. Another paper that looks at metrics is [33]. In this paper the area of research is creating labels for software. This is done to address the lack of focus on stakeholders,

when looking at energy consumption of software and to create awareness for them. This results in requirements for creation of sustainability labels and concluding that further work is needed for creating labels for software.

In [15] there is research into AC and DC power measurement methods as well as different model approaches for measuring power consumption, essentially looking at the hardware aspect of Energy-Aware Programming. The paper looks at several factors in the quantitative comparison of measurement techniques. Factors such as accuracy, overhead and sample rate are considered, and the results of the different techniques are compared to a calibrated, professional power analyzer. The paper found that AC data was reasonably accurate, APM and RAPL data was of varying quality, when the paper was published (2013), it is noted in the paper that these are expected to improve. Lastly DC measurements are concluded to be accurate with good temporal resolution.

Another paper that looks at hardware is [34]. In this paper the focus is on creating a high-performance computing infrastructure, which is heterogeneous that provides support in researching big data analytics and energy-aware high-performance computing. The reason behind this is that at the time of the papers publication there was a lack of both systems and tools that gave accurate, real-time and fine-grained measurements for power and energy. The paper presents a solution to these problems by designing the *Marcher* system and demonstrating its use. It is concluded that the system gives easy-to-use interfaces and tools, that allows developers to obtain the decomposed energy consumption of programs.

Beyond the software and hardware concerns of Energy-Aware Programming there is also research regarding Software Engineering itself. One such paper is [35], which investigates the level of concern developers have regarding energy consumption in their applications and what solutions they consider for improving consumption. The paper presents an empirical study on understanding developers understanding of energy consumption, utilizing *Stack Overflow* as the data source. The paper found that developers are aware of energy consumption problems and the questions regarding energy consumption were diverse and challenging. In regards to solutions to the problems and questions it was found that they were vague and/or flawed compared to state-of-the-art research.

In [36] Software Engineering is once again the area of focus. The paper looks at how evaluating energy efficiency of software applications is ad-hoc and seeks to establish a widely applicable model for how evaluation should be conducted. The paper's solution is a model called ME^3SA,

which gives the ability to measure and control energy efficiency in a similar manner to areas such as maintainability and performance. The model can identify hotspots for energy consumption alongside bottlenecks for energy efficiency, according to a case study conducted using the model.

In the paper [27] research of an energy measuring method was conducted. The paper looks at RAPL, to properly establish the strengths and weaknesses of utilizing it. It was found that, according to the observations in the paper, RAPL's energy readings were strongly correlated to reading from wall sockets, accurate, and had little to no performance overhead. This paper builds in part on [15] and finds that as software has matured, so has the reliability of RAPL. Part of the findings are also that there are issues with RAPL such as non-atomicity for the register updates and unpredictable timings.

Another paper [37] also looks at energy measurements, however instead of comparing and reviewing existing solutions there is instead a presentation of an alternative method. The paper presents a machine learning approach for predicting the energy consumption of software. As part of this it investigates determining how energy is consumed as while tools exist they do not look into where an application consumes energy. In this paper four different machine learning algorithms are used. The paper found that the error rate for predictions is less than 10%. Additionally, creating a model based on performance events, which can explain causes of energy consumption.

In [23], methodology and tools for measuring and benchmarking energy consumption are presented. In the paper, prior work is extended to account for multiple load levels and different combinations of workloads. This results in a power and performance measurement methodology as well as benchmarks and tools for the same purpose.

Next we look at the energy efficiency of different programming languages. The energy rankings of different languages are examined in [8] and [38], which is done by comparing the energy efficiency of the languages across several computing problems. Both papers found that while execution speed and energy consumption are correlated, a language does not necessarily consume less energy if it is faster. A part of research within this area looks upon the Energy Delay Product (EDP) of languages. In [39] the EDP of different languages are examined by testing common computing tasks across languages and platforms. EDP is a weighted function for the energy consumption and run-time performance of a sample of programming tasks. The result of [39] lead to the conclusion that compiled

languages perform better in general. Additionally, seeing that languages vary in performance depending on the task they are used for.

Comprehending the behaviour of different Java I/O API's is the area of research in [40]. The work is divided into two parts. First, an exploration of the behaviour of Java I/O APIs is performed. This exploration is followed by applying these APIs across applications utilizing inefficient APIs. The paper found that commonly used Java I/O APIs are not necessarily conducive to energy efficiency.

In [41] work regarding the recommendation of energy efficient collections in Java is presented. A tool called *CT+* is presented, which uses energy profiles of Java collections in conjunction with static analysis of the used collections to recommend alternatives that can improve energy consumption. The paper found that some widely used collections are not efficient and should be avoided when focusing on energy efficiency.

A similar area of research is automatically optimizing software for energy efficiency. In [42] an algorithm for post compiler optimization of energy usage called GOA is presented. The results of the paper are that there is an average reduction in energy usage of 20% when using GOA.

Another research area is the connection between maintainability and energy efficiency. In [43] the impact different energy changes have on maintainability is examined. This is achieved by looking at how maintainability changes for different repositories following an energy change commit. In [43] it is found that improving energy efficiency generally leads to a significant decrease in maintainability. Another similar study is conducted in [44], which looks at how common refactorings influence the energy efficiency of applications. This paper applies refactorings to applications and measures the energy efficiency of these applications before and after refactoring. In [44] it is found that refactorings have an impact on energy consumption. This area is also studied in [45], where the paper looks at how individual and combined refactorings impact the energy efficiency of an application. To accomplish this a tool called *Chimera* was developed, which automates the process of identifying refactoring candidates and application of refactorings. The paper found that individual refactorings result in a positive impact on energy efficiency, whereas combining refactorings generally increases energy savings, but some combinations result in decreased energy efficiency.

2.2.2 Energy Consumption & Language Constructs

Given that we have an overview of parts of the research in Energy-Aware Programming, we move into a deeper exploration of energy consumption and language constructs. Most of the previous research and experiments have been conducted in Java, specifically in [4], [46], and [11].

As of December 2021, not much known research besides these three papers have been done into language constructs. However, existing research does give insight into how some language constructs are more energy-efficient than others, e.g. choice of data type and different ways of concatenating strings.

Furthermore, [9] presents comparisons between different programming paradigms in C# and F# with regards to energy consumption for some algorithms. In [9] they do not present the results for specific language constructs.

Energy Consumption in Java: An Early Experience

In [4] multiple different results regarding the energy consumption of language constructs in Java are presented. The paper presents measurements of the energy that different language constructs consume and compares them to the alternatives.

There are many findings in this paper, the main ones include:

- primitive data types are more energy-efficient than wrapper classes,
- static variables causing 50% increase in energy consumption compared to local variables,
- static variables causing 60% increase in execution time compared to local variables,
- short circuit operators having better energy efficiency if the first case is the most common one,
- string concatenation being more energy-efficient than `StringBuilder` and `StringBuffer` append methods,
- method calls in statements like for loops does not always have increased overhead, and
- try-catch blocks have no cost if no exceptions are thrown.

We consider the most surprising of these results to be that local variables are more efficient than static variables, method calls not always having overhead, and try-catch blocks having no cost if no exceptions are thrown.

These findings are the results of comparing intra-language constructs and can serve as a base for inter-language comparisons with languages that share the same characteristics e.g. C#, to test whether the same findings apply. We delve into [4] later in this chapter when we look at the papers concerning language constructs and energy consumption in Java.

Improving Energy Consumption Of Java Programs

In [11], the area of energy consumption in Java is explored further, with an added focus on how the energy consumption can be improved.

Many of the experiments in [11] arrive at the same conclusion as [4], however, there are some key differences, for example, the native concatenation operator for strings is shown to be significantly slower than the different `append` methods and the `concat` method on a string, which is contradictory to what was shown in [4]. Furthermore, in [11], a high linear correlation between energy consumption and execution time is found, with a correlation of at least 96% for the single-threaded workloads tested. Different Java command-line arguments are compared with regards to energy efficiency and find that the argument *UseG1GC* causes the highest energy efficiency. Open JDK is compared to Oracle JDK. The paper finds that Oracle JDK generally outperforms Open JDK with regards to energy efficiency. As with the previous paper, we choose to delve into [11] when we look at the papers concerning language constructs and energy consumption.

Inheritance versus Delegation - which is more energy efficient?

In [46] there is an investigation of whether `Inheritance` or `Delegation` consumes the most energy. Preliminary experiments in Java are performed using code snippets from Fowler's refactoring textbook [47].

The experiments are run in interpreted mode, meaning that the bytecode is not compiled at run-time, but instead interpreted as it is executed. The reason the experiments run in interpreted mode is to avoid optimizations from the compiler. The experiments are run 25 times each to reduce potential outliers. In between each experiment, a pause of 30 seconds is

added to maintain continuity and reduce energy consumption to idle levels before the start of the next experiment. The benchmark method is executed 100 million times per experiment, allowing the program to run for longer periods of time, during which power readings are collected. Longer run-times are necessary as the Watts Up Pro power meter is used to measure power consumption, which has a sampling rate of one second, as described in Section 2.1.1.

The results in [46] show that the Inheritance version has a 77,93% reduction in run-time compared to the Delegation version, and the Inheritance version consumes 81,36% less energy than the Delegation version.

These results show that refactoring can have an impact on the energy consumption in programs.

The Influence of Programming Paradigms on Energy Consumption

In [9] results regarding different programming paradigms and how well they perform with regards to energy consumption are presented. The experiments are for a simple neural network and 10 algorithms across paradigms, which are chosen by the authors and are from Rosetta's Code repository [48].

A dedicated computer is used for the project since they state it makes their results more reliable. The main results of the report include that Procedural Programming (PP) in general, is faster than Object-Oriented Programming (OOP) which, mainly, is faster than Functional Programming (FP) with a few exceptions. The exceptions include:

- Dijkstra's Shortest Path is most efficient in OOP in F#,
- Playing Cards is more efficient in FP compared to OOP in F#,
- Game of Life is more efficient in FP compared to OOP in F#,
- Playfair Cipher is most efficient in OOP in C#, and
- K-means has mixed results in F# and is most efficient in FP in C#.

We consider [9]'s use of a dedicated computer relevant to our project, as it gives an approach for how we can achieve reliable results. Furthermore, it shows that FP and OOP could use optimization with regards to most of these algorithms.

Lastly, a significant contribution of this project was that a measurement library for C#, interfacing with RAPL was created.

2.2.3 Energy Optimization Tools

Now that we have seen examples of research into energy consumption and language constructs we move on to the next part of related work. Here we look at tools that are made for measuring and/or optimizing programs with regards to energy consumption, where most of these tools have been created for Java.

Specifically, we look at the papers [49], [50] and [51], which present different tools used for optimizing programs with regards to energy consumption. The first two papers present tools made for Java while the third paper presents a tool created for the Integrated Development Environment (IDE) *Visual Studio Code*, specifically focusing on C#.

jStanley: placing a green thumb on Java collections

In [49] a tool called *jStanley* is presented. *jStanley* automatically optimizes Java programs with regards to energy consumption.

The tool is created as an extension to the IDE *Eclipse*, it focuses specifically on collections. The tool will give suggestions on what type of collection should be used in different circumstances to improve energy efficiency. Additionally, it provides functionality to focus on execution time.

The tool provides a foundation to compare against when creating similar tools. The paper concludes that the tool optimizes energy consumption by 2%-17% and execution time by 2%-13%. Furthermore, the tool can be improved by taking other language constructs and libraries into account.

SEEDS: a software engineer's energy-optimization decision support framework

In [50] a framework called SEEDS API, which optimizes Java applications is presented. Specifically, Java applications using the Collections API. The framework creates a new version of the application with changes that are more energy efficient. The tool can do these improvements automatically, hidden from the developer.

The paper found that the tool optimizes energy consumption by 2%-17% when improving applications using only the collections API.

The SEEDS API is the first known framework [50, p. 512], that helps software engineers to make energy-conscious choices during the construction of applications, and according to the paper capable of improving energy us-

age by up to 17%. Furthermore, the tool provides a good basis for further investigation into tools for software engineering decisions.

IDE Extension for Reasoning About Energy Consumption

In [51] an IDE extension for *Visual Studio Code*, which helps developers reason about the energy consumption of their C# programs, is presented. The extension helps by giving the developers an estimate of how much energy their code consumes but does not offer insight into improving power consumption.

The extension uses both static as well as dynamic analysis to determine the energy consumption of a piece of code. The static analysis consists of machine learning, as well as creating an energy model, while the dynamic analysis executes the code and measures the energy consumption using RAPL.

The IDE extension uses dynamic analysis as ground truth towards testing static analysis, which means the results for the machine learning is relative to the results from RAPL. The energy model predicts at worst -100,56% to 74,98% from the ground truth, while the median prediction is -79,22% compared to the ground truth. For the five machine learning approaches, the one which provides the best results is Random Forest which at worst predicts -7,49% to 9,19% from the ground truth, and for median prediction, it is 1,06% above the ground truth.

2.2.4 Existing Java Experiments

As the last part of our look into related work, we delve into the two papers [4] and [11]. As established earlier these papers concern themselves with language constructs in Java and as such, they are closely related to the area of study of this project. Furthermore, the papers give us an insight into how different constructs can be grouped, how they can be tested, and what results to expect.

In Table 2.1 [4] and [11]'s groupings of their constructs are seen. Important to note is that the groupings are not the same between the papers.

Energy Consumption in Java: An Early Experience [4]	**Improving Energy Consumption Of Java Programs [11]**
Variables	Variables
Strings	Strings
Exceptions	Exceptions
Arrays	Arrays
Short circuit operators	Operators
Loops	Control Statements
	Objects
	Threads

Table 2.1: Papers [4] and [11] with their groups of constructs which they explore.

Variables are used to store information. In Java, there are four different kinds of variables; instance variables (non-static fields), class variables (static fields), local variables, and parameters [52].

In [4] there are comparisons between the energy consumption of the primitive data types, `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`, as well as comparisons between local and static variables. While in [11] there are comparisons between the energy consumption of all the same primitive data types except for `boolean`. In [11] additionally investigates the difference between static and local variables in Java.

The results in [11] show that `int` consumes the least amount of energy and double consumes the most. The results in [11] for the static and local variable comparison shows that the static variables consume up to 17.700% more energy than local variables. In [4] the results for variables show that `long` is the most efficient primitive data type. For static and local variables the results in [4] show that static variables raise energy usage by 50% compared to local variables.

Strings in Java are represented by a class with the same name. The `String` class represents an array of characters [53].

In both [4] and [11] there are comparisons between the energy consumption of the `String` concatenation operator (+), the `String` append methods from `String-Buffer`, and `StringBuilder`.

In [4] it is concluded that the `String` concatenation operator was better in regards to energy efficiency and speed, where `StringBuilder` took 10% more time than the `String` concatenation operator. For energy consump-

tion the `StringBuilder` consumed less energy in the beginning, but over time the energy consumption would increase to a similar cost compared to `StringBuffer` and the `String` concatenation operator.

Where in [11] it is found that using the `String` concatenation operator has a higher energy consumption by up to 148.069% compared to using the append methods from `String-Buffer`, and `StringBuilder`.

Exceptions in Java consists of two categories, namely checked and unchecked exceptions. Checked exceptions must be handled by wrapping the code in a try/catch block for the code to compile. Unchecked exceptions are also called run-time exceptions and do not need to be inside a try/catch block for the code to compile [54].

Both [4] and [11] investigates the energy consumption of a try-catch block in a program that does not throw any exceptions. Furthermore, they compare the difference between having a try-catch block in a loop versus having a try-catch block outside a loop.

In [4] the results show that try-catch blocks have almost no energy impact when no exception is thrown. Furthermore, it was found that using an if statement for error checking caused higher energy consumption.

However, in [11] the results show that throwing an exception consumes up to an additional 3,29% energy than when no exceptions are thrown. Furthermore, [11] shows that an exception outside of a loop consumes up to 24% more energy than having the exception inside a loop.

Arrays in Java is a container that can hold a fixed amount of the same type. The length of an array is set upon creating the array and can thereafter not be changed [55].

In [4] it is explored if there is a difference in energy consumption between using a variable or an array with a single element. In contrast to this in [11] copying arrays using different methods, as well as two-dimensional array traversal, are explored.

In [4] the results show that `Arrays` consume 33% more energy than local variables. In [11] the results show that `Arrays.copyOf()` consumes the most energy, increasing consumption by 14% compared to `System.arraycopy()` which consumes the least amount of energy.

Short Circuit Operators in Java refers to the use of the conditional operators `OR` (`||`) and `AND` (&&) to perform a conditional operation on a boolean expression. The conditional operators perform a short-circuit action, meaning that the later operands are only evaluated if needed [56].

In [4] the short circuit operators `OR` (`||`) and `AND` (&&) are examined. The paper tests the difference of having the evaluation for the first operand being enough to terminate the statement versus having the last operand terminate the statement. An example is

$$TRUE||FALSE||FALSE$$

versus

$$FALSE||FALSE||TRUE$$

The results in [4] show that putting the most common cases first in a short circuit operator saves up to 10% of the energy consumption.

Loops are used to execute statements a number of times until a condition is fulfilled. Examples of loops are for, while, and do-while [57].

In their analysis of loops, [4] first changes the initialization variable to either `int`, `long`, or `double` in a for loop, while they make an addition assignment inside the loop. In addition, the paper tests two different ways of initializing the termination expression, one using a variable and the other using a method call.

The results in [4] show that `int` is the most energy efficient initialization variable in a for loop. The result for which termination expressions consumes the most energy shows that using a method call consumes the least amount of energy.

Operators are symbols that perform different operations on one, two, or three operands, and then return a result [58].

In [11] binary operators are looked at. The paper tests the operators by performing arithmetic operations using `long` and `double` operands, and storing the values in the same `long` and `double` variables. Furthermore, the paper compares compound assignments with the alternate approach where the operands is first evaluated on the right side of the statement and then assigned to the variable on the left. The paper also compares compound assignments with post-increment, pre-increment, post-decrement, and pre-decrement. In addition [11] investigates short circuit operators the same way [4] does in **Short Circuit Operators**.

In [11] the results for binary operators show that division and modulus consume the most energy for long operands, with modulus consuming 1.620% more energy than the addition operator which is the operator that consumes the least energy. For double, the results show the same behavior,

except that division consumes less energy than the other operators. For compound assignments compared with the normal, it is found that it does not matter which is used, as they consume the same amount of energy. The findings in [11] show that the short circuit that terminates the statement after the last operand is evaluated increases energy consumption up to 100% compared to statements that terminate after the first operand.

Control Statements are used to control the flow of execution for a program. Controlling the flow is done through looping and branching which gives the opportunity to conditionally execute specific blocks of the program [59].

In [11] a conditional operator called ternary operator is looked at. This is an if-then-else statement written in one line. The tests are done on two different system component setups, one Intel Fog Node configuration, and the other a laptop configuration. The paper compares the ternary operator with an if-then-else statement. The paper investigates the three loop statements, `for`, `while`, and `do-while`, as well as comparing the `for` statement with the enhanced `for` statement. In [11] loops are looked at, likewise with [4] which was described in **Loops**.

In [11] the results for the ternary operator show that for the Intel Fog Node setup there is less than 1% difference in energy consumption for the ternary operator and the if-then-else, while on the laptop setup the ternary operator consumes up to 37% more energy. For the three loop statements, it is found that they all consume the same amount of energy. The result for the `for` and enhanced `for` statement show that they consume the same amount of energy. In regards to which iteration variable of a for loop consumed the least energy, it is found that `int` consumes the least and `double` consumes the most, with an increase of up to 573% compared to `int`. When looking at which termination expression consumes the most energy, the results show that method termination consumes up to 3% additional energy than the variable case which is contradictory to what [4] found in **Loops**.

Objects in [11] are looked at only in regards to boxing primitive types, which means turning primitives into their wrapper class counterparts, for example, `int` would be boxed into an Integer object [60, 61].

In [11] the data type wrapper classes are tested to see which are more expensive, namely `Integer`, `Long`, `Float`, `Double`.

The results in [11] show how the `Integer` consumes the least amount of energy, while `Double` consumes the most, raising consumption up to 115% compared to `Integer`.

Threads are a part of the execution of a program. The Java Virtual Machine (JVM) allows to have multiple threads running concurrently. In Java there are two ways to create threads, one is to declare a class a subclass of the `Thread` class, the other is to create a class that implements the `Runnable` interface [62].

In [11] the energy differences between extending the class Thread or extending the interface `Runnable` are looked at.

The results for threads in [11] show that the `Thread` class and the `Runnable` interface consume the same amount of energy.

2.2.5 Summary

There is a large range of research areas within Energy-Aware Programming. We have looked at areas from metrics, hardware, energy measurement, methodology, language rankings, specific API's and collections. We followed this by looking at more non-functional concerns of how maintainability may be influenced by energy aware changes to code. One should note that other areas also fall under Energy-Aware Programming and what is seen in Section 2.2 is not an exhaustive list of all related work.

Looking at the research more closely related to the project's area of concern, there is research concerning the Java programming language, both with regards to measurements of different language constructs and with tools created to optimize energy consumption.

Furthermore, there is a limited amount of research within the C# programming language, with the research mainly focusing on algorithms with regards to measurement and reasoning about energy consumption in an entire C# program. Part of this research also results in an interface with RAPL in C#, thereby allowing for energy measurements, without the use of external hardware.

We observe that even though there is research into language constructs, the amount of research is limited, considering the research being from only two papers. Additionally, these papers are only concerned with the Java programming language, and as such knowledge of language construct impact on energy consumption is very limited, also keeping in mind that the papers [4] and [11] are not exhaustive lists of every language construct. Additionally, the work presented in these papers show that some language constructs are more efficient than others, giving us reason to extend this research area to expand this knowledge. Another observation we have made through this look at related work is the existence of an interface for RAPL

in C#. This is interesting because energy consumption research mainly focuses on Java, and as such this interface allows us to more easily conduct research into language constructs in C#.

Seeing as a foundation exists for measuring programs in C# and there is a lack of research into language constructs within it, we choose to research language constructs in C#.

Chapter 3

Language Constructs

Language constructs are the building blocks for programs in any programming language. In this chapter, we present the language constructs we test and the motivation behind them. Furthermore, we explore these language constructs, what they are, and how they are used. Lastly, we explain how we group the language constructs for energy comparison. We use [4] and [11] as a starting point, where we look at constructs in C# that are equivalent to those of Java. We choose these papers as a starting point as they are directly related to researching language constructs.

3.1 Selection of Constructs

We group the different constructs according to what we want to compare. The main idea behind each group is that the outcome of code that uses an element from a group should be equivalent no matter which element in that group is used. For example, you can exchange for loops with a while loop without changing the outcome of the code. There are a few exceptions to this, as there will always be a different software state when using a `short` instead of `int`. That being said, the only difference here is that there is a `short` variable instead of an `int` variable in that state and as they serve the same purpose, they are considered to be the same group.

Grouping	Constructs
Primitive types	bool, byte, sbyte, char, decimal, double, float, int, uint, nint, nuint, long, ulong, short and ushort
Operations	+, +=, -, -=, *, *=, /, /=, %, %=, ++, - -, <, <=, >, >=, &&, ||, », »=, «, «=, &, &=, |, |=, , ^, ^=, ==, != and !
Variables	instance, static variables, local variables, static properties, instance properties and formal parameters
Selection	if and switch
Loops	do while, while, for, foreach, goto, recursion and Language Integrated Query (LINQ)
Collections	collections and arrays
String Concatenation	+, += , string builder, Interpolated strings, string.Format, string.Concat and string.Join
Boxing	Boolean, Byte, SByte, Char, Decimal, Double, Int16, UInt16, Int32, UInt32, Int64, UInt64, and Single
Invocation	instanced method, static method, reflection, delegate and function pointer
Field Mutability	mutable fields, init only properties, get only properties, constants and read only
Instance Variables	properties, property with backing field, method getters/setters and fields
Objects	class, struct and record
Inheritance	abstract, interface and no inheritance
Generics	generics and non-generic
Jumps	break, continue, return, throw and goto
Exceptions	try-catch, try-finally and try-catch-finally

Table 3.1: The groupings and the language constructs within the groupings.

In Table 3.1 we list the constructs we test along with the grouping they belong to, we now discuss why we choose these. Important to note is that not all of these groupings give unexpected or interesting results, and therefore the results of the language constructs with unsurprising results are present in Section A.3.

Primitives & Operators

We look at primitive types and operators and their energy usage similar to what was done in [4]. In this project we compare each of the primitive types

from C#, doing different kinds of operations on these types e.g. addition, subtraction, division, multiplication, and modulo. Additionally, we look at boolean operators and their energy consumption, e.g. logical negation, conditional logical AND and conditional logical OR. Besides this, we also look at bitwise and shift operators and their energy consumption, e.g. bitwise logical AND and bitwise logical OR. This comparison tells us if it is better to use a smaller data type like `int` instead of a larger data type like `long` if it can be changed without altering the outcome of the code, or if it is better to use `long`. The same knowledge can be acquired for `float` and `double` and other data types. Lastly, it lets us know how long each of the operations will take and how much energy they consume.

Variables

[4] finds that static variables use 50% more energy compared to local variables. Because of these results, we explore if the same holds true for C#. To explore C# we test the energy consumption of instance variables, static variables, local variables, static properties, instance properties and formal parameters. Exploring this gives an insight into if one of these definition types should be preferred where possible.

Selection

In [11] it is found that ternary operators increase energy consumption by up to 37% in comparison to a semantically equivalent if-then-else statement. Thereby giving us a reason to look into if there is a difference between the different selection statements in C#.

Looping

In [11] it is found that `for`, `while` and `do-while` loop statements consume a similar amount of energy, and the `for` loop consumes the same amount of energy as the enhanced `for` loop. We want to look into if the same is true in C# or if there is a difference in energy consumption between the iteration statements. Additionally, we want to compare these loop statements with recursion, goto, and LINQ, to see if these could be used as energy-saving alternatives. LINQ is a data querying API, which is used to work with data access from objects, relational databases, XML, and others [63].

Collections

As seen in [49] and [50] choosing an appropriate collection for the application/code can improve energy consumption. These papers looks into collections specifically in the Java language which gives a way for us to explore this area in C# to get knowledge regarding which comparable collections have better energy efficiency. Therefore, we test collections and arrays with the purpose of finding differences between their energy consumption.

String Concatenation

Previous work [4] and [11] test different methods of concatenating strings in Java, and found differences in both run-time and energy consumption. C# has similar constructs so we want to know how operator concatenation (+,+=), string builder, interpolated strings, `string.Format`, `string.Concat` and `string.Join` affect energy consumption. Experiments into these would show the differences between these approaches in the form of how they affect energy consumption, which could be used to inform developers on which construct to use.

Boxing

In [11] the energy cost of boxing primitive types is tested and discussed. For example, when a primitive is added to a collection it is turned from a value type into an object, this is done to be able to reference these primitive types [64]. We want to determine the effect that boxing has on energy efficiency in C# and if this effect is similar to what is discovered in [11].

Exceptions

In [4] and [11] they find that using a `try`-block costs nearly nothing with regards to energy efficiency and run-time when no `exception` occurs. When an `exception` occurs, it is shown that it costs between 2,46% and 3,29% more than when no `exception` is thrown.

Because of this, we want to test if the same holds in C#. Here it is possible to control the flow of the program, despite errors occurring. This can be done using `try-catch`, `try-finally` and `try-catch-finally` blocks [65, 66, 67].

Getting knowledge regarding the energy consumption of these blocks gives insight into how one should handle exceptions if programming with energy efficiency in mind.

Other Language Constructs

We have now covered the groups from [11, 4] and look into the other language constructs groups from Table 3.1, that were not studied in these papers.

Invocation

In C# you are offered several mechanisms for invocation and investigation into these will show the energy consumption and run-time of the different constructs. Comparing static method invocations and instanced method invocations, could help in deciding if it is better to have static methods or if they are equal in energy consumption. C# has a construct called `Delegate` which represents a reference to a method, which allows C# to express Higher Order Functions (HOFs) [68]. `Delegates` can be used in the form of the class `Action` which has no return type and the class `Func` which does have a return type [69, 70]. In C# 9 they introduced function pointers which allows for increased control over the calling convention [68]. Comparing function pointers and delegates along with its other forms will inform us of the cost of using these abstractions. C# also allows for method invocation using reflection, we want to see what the energy and run-time costs for using reflection are [71].

Field Mutability

C# offers different mechanisms for field mutability which we want to test and see if it has an effect on the energy use. We look at multiple kinds of field mutability, init only, get only, mutable fields, constant fields, and readonly fields. Constants are known at compile time and do not change after they have been declared and initialized. The init gives the possibility to mutate the members of an immutable object during the construction of the object. Get only properties are achieved by only making getters, and therefore being unable to change values for the fields. Readonly value types are also immutable, however a field that is a readonly reference type always refer to the same object, that object is not immutable. We want to explore

if one type is better in regards to energy efficiency than the other, and therefore should be preferred when programming with energy saving in mind [72, 73, 74].

Instance Variables

C# offers properties which is a way of creating getters and setters when declaring data members [75]. Properties act like a field with getters and setters, the compiler will generate a backing field to represent the data itself [75]. We want to know if there is a difference between C# properties, a method implemented getters/setters, and public fields. With the knowledge of differences between the properties, getters/setters, and fields, we can determine which of these methods of declaring data members are best for energy consumption. We can specifically utilize this when comparing method implemented getters/setters to C# properties. Furthermore, we could learn what effect getters and setters have on energy consumption compared to fields defined without getters or setters.

Objects

In contrast to Java, C# offers three different ways of defining objects: Classes, structs, and records. All three can be used for the same purpose.

A class in C# functions the same as a class in Java, meaning they are reference types [76]. A struct in C# functions the same way as classes, except that structs are a value type [77]. A record is a bit different as it, by default, is an immutable reference type, meaning it cannot be changed after creation. Records can be made mutable, meaning the difference between classes and records are minor, where the main difference is the built-in behaviors for records such as value equality [78].

We want to know whether using one of these is more efficient with regards to energy consumption than using another one. Specifically, we want to compare classes, structs, immutable records and mutable records.

Inheritance

In C# there are two main ways to create inheritance [79]. It can be done with the help of interfaces and abstract classes. Furthermore, it is possible to create the same functionality in a subclass without inheritance, by implementing all the functions by hand.

The main difference between interfaces and abstract classes is that it is possible to implement multiple interfaces, but only one abstract class [80, 81].

A large difference between Java interfaces and C# interfaces is that interfaces in C# can implement default behavior for methods, which can be overwritten like in abstract classes [82].

The differences between these approaches could have implications on how efficient each of them are in regards to energy consumption and runtime. Based on how they can be used for the same purpose we want to explore them to determine if one is preferable to another.

Generics

In C# it is possible to create generic classes. making it possible to defer the types of one or more parameters until the class is instantiated [83]. These generic types are substituted at run-time, which means that there could be a performance and/or energy penalty to using a generic class as opposed to a fully parameterized implementation of a similar class, which we want to explore.

Jump statements

In C# there exists four jump statements that can be used to transfer control from different points in a program, the statements are `break`, `continue`, `return` and `goto`.

The `break` statement is used to terminate a surrounding loop or an associated statement. Control is then be passed to the next statement. If the `break` statement is placed in a nested loop it will only terminate the loop which contains it [84].

The `continue` statement passes control to the next iteration of a surrounding loop structure, skipping statements that follow it [85].

`return` is used for termination of an accompanying method and passes control back to the caller and can return an optional value. For methods that are of type `void` the `return` statement is not required [86].

A `goto` statement is used to transfer control to a labeled statement, the label is placed just before the statement to which control is transferred [87].

Getting knowledge regarding the energy consumption of these different jump statements gives insight into whether a specific type of jump statement should be used over another.

3.2 Summary

We choose to investigate 16 different groups of constructs in the C# language. These are seen in Table 3.1. Seven of the groups are based on related work, which found that some Java constructs have differences in energy consumption for code that has equivalent results. Beyond those found in related work, we look at nine other groupings. We investigate if C# constructs, like the Java constructs, have different energy consumption within the different groupings.

Chapter 4

Experiment Design

In this chapter, we set up our goals for the experiments. We set up the goals to understand what our experiments are designed to determine, as well as how to best construct them to fulfill these goals. Afterwards, we present our design choices for the experiments and what our guidelines for the experiments are. As a part of this chapter, we look at the differences between microbenchmarks and macrobenchmarks to choose the type of benchmark to use. Additionally, we describe the approach we choose for measuring the results of our experiments. At the end, we discuss the threats to validity that impact the results.

4.1 Goals

The overall goal of this study is to determine whether different language constructs in C# use more, the same, or less energy than other options that are similar, as shown in Section 3.1. Related work (Seen in Section 2.2) has uncovered differences in other languages. To formalize our goal for the experiments, we create the following hypothesis:

"There is a difference in energy consumption between language constructs that achieve similar outcome in C#".

To achieve the goal of comparing the energy consumption of language constructs, several variables must be taken into account.

A suite of benchmarks needs to be created, which provides its own set of challenges. For example, how many times must an experiment be executed to get a significant result, and what is the correct approach to testing different constructs.

To make benchmarks we follow the criteria described in [88]. These being Relevance, Reproducibility, Fairness, Verifiability and Usability.

Relevance

Benchmarks may be highly relevant in one scenario and of minimal relevance for others. Thus it is important to determine the intended use of the benchmark and design the benchmark so that it is relevant in the targeted areas [88].

Reproducibility

Reproducibility is the capability of a benchmark to produce consistent results in a specific test environment. This involves both result consistency between different runs and the ability for another tester to independently reproduce the results on another system. To do this it is important to include a description of the test environment, including both hardware and software components as well as configuration options [88].

Fairness

Fairness is important to allow different systems to compete on their merits without artificial constraints. Benchmarks have some degree of artificiality as it is often necessary to place constraints on test environments as to avoid unrealistic configurations which may take advantage of the simple nature of the benchmark [88].

Verifiability

The results of experiments must be verifiable, as such, good benchmarks perform some amount of self-validation to ensure that an experiment is running as expected. An example of self-validation is that the experiment verifies that the result of running the experiment is correct, as an incorrect result means that any measurement would be invalid. One way to improve verifiability is to include extra details in the results that may not be part of the results we are measuring. For example, if measuring energy consumption, we could include the temperature of the CPU as it may have an effect on energy consumption, but is not part of the actual result. Doing so can

help spot inconsistencies in the resulting data which can raise questions of the validity of the data [88].

Usability

Usability concerns the ease of use of the created benchmarks. Self-validation is an important feature for ease of use of benchmarks. With self-validating workloads the developer does not have to concern themself whether the workload is running properly [88].

4.2 Microbenchmarks vs Macrobenchmarks

A microbenchmark is a benchmark that tests a small amount of code, sometimes only a single instruction [89]. There are different sizes of microbenchmarks, for example an algorithm is considered a microbenchmark, however in our case we focus on smaller microbenchmarks, which look at specific language constructs. When using microbenchmarks you are testing well defined variables, examples of such are performance, time elapsed, resource consumption and energy [90]. A macrobenchmark is the opposite, which tests a larger amount of code, sometimes an entire program [89]. Even though the benchmark is at an application level, testing revolves around the same variables [90].

The advantage of utilizing microbenchmarks is that we can test something specific while keeping all other variables constant. In our case this means that individual language constructs can be measured while almost everything else is kept constant. This makes it possible to figure out what language constructs have greater efficiency than other constructs, and can make it possible to optimize applications written in the language tested [89].

A disadvantage of utilizing microbenchmarks is that they may not be representative of an entire program, as lesser utilized constructs may play a big part of a microbenchmark but may be negligible in a larger program [89]. Furthermore, generalisation of microbenchmarks to a larger program may not behave as expected, this is explored further in Section 4.6.3.

The advantage of utilizing macrobenchmarks is that they are representative of an entire program, which means that macrobenchmarks can be used to measure how efficient an entire program is. Furthermore, small changes in external variables, for example CPU temperature, have less effect on a macrobenchmark compared to a microbenchmark [89].

The main disadvantage of utilizing macrobenchmarks is that small changes in programs are difficult to capture, meaning it is difficult to know if a small change has made any impact on overall efficiency [89].

Overall, microbenchmarks are good where macrobenchmarks are bad and vice versa. In our case, we want to find the effects of specific language constructs with every other variable as constant as possible, so microbenchmarks are chosen. We leave it for future work to see if the same effects exist in macrobenchmarks.

4.3 Choice of Benchmarks

In this section, we explore different benchmark problems used in related work to choose which benchmarks to use in our project, that uphold the criteria detailed in 4.1. Previous work [50] creates 13 microbenchmark problems to test collections of the Java language. In [49] seven Java projects which use the Java Collections Framework are chosen. These are chosen from a free open-source software platform called Sourceforge [91]. We are not able to apply the benchmarks from [50] or [49], because they are created for Java, and specifically focus on Java Collections.

Previous work [51] scrapes different online repositories for C# programs, these repositories are Rosetta Code [48] and Computer Language Benchmarks Game [92]. Additionally, learning platforms of Sanfoundry Global Education and Learning [93] and Include Help [94] are scraped. [51] discovers 1438 different C# programs of which 147 are chosen. Each of these different studies use these scraped programs and repositories as their benchmarks. Using these allows them to perform their studies on real world examples, making their results a reflection of how their work would affect real world applications.

Standard Performance Evaluation Corporation (SPEC) has created several benchmarks, that includes a SPEC power benchmark suite, which purpose is to measure the power and performance characteristics of server-class computer equipment. Their power and performance suite is called SPECpower_ssj 2008. Access to the suite costs 1600 dollars [95]. As the suite is a proprietary product we decide not to use it [23]. Our choice is based on our goals for reproducibility.

Based on the previous work of [50], [49] and [51] we decide to create our own benchmark tests. This choice is made because none of the existing suites are directly usable for the type of testing we perform. Instead we

are able to use the experience gained from the previous work to make tests, where we can mitigate some of the threats to validity described in these previous works.

4.4 Measurement Approach

In this section we determine the approach for measuring our experiments, including the performance parameters and how they are obtained. We choose to collect energy consumption and execution time for our experiments.

Part of this is to determine the tools used for measuring the energy consumption of different constructs. In Section 2.1 we looked at a number of different methods for measuring the power consumption of a computer system. We found that the software-based measuring methods allowed for finer granularity in regards to the frequency of measurements, than the hardware-based solutions. For the experiments we want to measure both the execution time and energy consumption of the different language constructs and collections, as such we choose to utilize a software-based power measuring approach.

In Section 2.1 we looked at Running Average Power Limit (RAPL) and found that previous studies have found that it is an accurate approach to measuring power consumption. Based on these findings we choose to use RAPL for preliminary testing, both to test the tool itself and how further experiments will function. The results of the preliminary experiments can be seen in the appendix in Section A.1. Based on the preliminary testing we have elected to continue using RAPL. Seeing as the benchmarks are not in need of specialized hardware outside of the memory and processor, the domains covered by RAPL are sufficient for the data we capture. Additionally we choose to use the default sample rate of RAPL, which is 1 ms.

4.5 Experiment Guidelines

In this section we describe the guidelines for how to set up benchmark tests. We create these guidelines to ensure that the benchmarks are consistent. By having consistent benchmarks it allows for future research to follow the style we have created.

Number of Measurements

As modern computer systems are practically non-deterministic due to the many unpredictable variables when running and measuring programs, we need to run a program multiple times to get a result that has any statistical significance [96].

To get a representative sample of measurements, we use the approach recommended in [96]. We do this to have a methodology behind creating and running usable and reliable benchmarks.

Specifically, [96] recommends that we should:

- use a large number of iterations,
- use a number of iterations large enough to run for at least 0,25 seconds,
- save the results inside the benchmark to a dummy variable so a compiler does not optimize it away, and
- run the entire benchmark at least 10 times so we can compute standard deviation.

Instead of running the entire benchmark 10 times, we utilize a formula presented in [97] to calculate how many runs we need to be 95% confident that one benchmark is significantly different than another benchmark. As preliminary testing (seen in Section A.1.7), together with similar experiments [98, 9] show that the distribution approximates a normal distribution, we can utilize the formula seen in Equation 4.1. The 95% confidence (p-value of 0.05) is chosen as it is considered significant [99].

$$n = (\frac{z_{\alpha/2} * \sigma}{E})^2 \tag{4.1}$$

Where n is the number of samples needed to get 95% confidence, z is the z-score which can be looked up in a z-table or calculated by taking

the distance between the input and the mean, and dividing the result with the standard deviation. α is the p-value (1-confidence), σ is the standard deviation and E is the desired margin of error. In our case the margin of error is 0,5%, as this value has been used in similar work [9]. Because it is a percentage value instead of an absolute value of a specific metric, we need to multiply it with the mean of the sample population, meaning the formula ends up looking as seen in Equation 4.2.

$$n = \lceil (\frac{z_{\alpha/2} * \sigma}{r * \mu})^2 \rceil \tag{4.2}$$

Where r is the relative margin of error, 0.005, and μ is the mean of the sample population. Utilizing this formula makes it possible to know how many times we need to run a benchmark to get a representative sample. The result needs to be rounded up, as the margin of error will be larger than the desired if it is rounded down.

The number of samples needed is recalculated after each benchmark run, as the mean is not known before running the benchmarks. Once the number of runs exceeds the number of samples needed, an acceptable result is reached and a conclusion can be made.

Code Setup

When setting up code to compare different constructs in the same group, as seen in Chapter 3, the only difference between each of the code snippets should be the specific language construct being tested. This ensures that we only test the specific language constructs instead of any other small change in the code.

Furthermore, a `for` loop is used around the measurement to create multiple iteration. Results of preliminary testing (seen in Section A.1) have shown that utilizing a `for` or a `while` loop is less varying than unrolling the loop, even when utilizing ReadyToRun (R2R) compilation [100]. Furthermore, preliminary testing has shown that the difference between using a `for` loop and a `while` loop is negligible (seen in Section A.1.3).

A timer and power measurement is initialized before the `for` loop to measure how efficient the experiment is with regards to run-time and energy consumption. The timer and power measurement ends right after the `for` loop. The timer is started before the power measurement and ended after the power measurement to ensure the overhead of starting and ending the timer is not captured in the power measurement.

The number of loop iterations is saved after the `for` loop so the measurements can be compared with other language constructs, regardless of whether the number of iterations are the same. Comparing the measurements regardless of iterations is done by dividing the measurement numbers with the number of iterations and then multiplying it by 1.000.000 to get numbers that are normalized, and thereby comparable.

Furthermore, as we also follow the tips presented in [96], we ensure that loops are not optimized away by the compiler. Specifically, we do this by saving the results inside the loop to a dummy variable. The resulting state of the dummy variable is saved to make it possible to check if the experiment has run correctly.

4.6 Threats to Validity

To be able to reflect on the results from the experiments, we look at the threats to validity. These are divided into 4 different categories, *Construct Validity*, *Internal Validity*, *External Validity* and *Reliability*. The categories are inspired by [101].

4.6.1 Construct Validity

Construct validity refers to whether an experiment measures what it claims. The goal of the experiments is to measure the energy consumption of distinct language constructs in C# as described in Section 4.1. This means we have to make sure we are actually measuring the constructs we are expecting. For example, we need to make sure the code is not optimized away by the compiler. To make sure we are measuring the correct construct we can make variable-dependent loops, as well as check the Intermediate Language (IL) and machine-code generated by the compiler. If the code is shown to have been optimized away by the compiler, we tweak the experiments until the code is not optimized away by the compiler, to get a usable result.

4.6.2 Internal Validity

Internal validity refers to whether other factors than the variables used in the experiments influence the results.

There are several different internal threats to validity, these include the temperature of the CPU and testing environment, OS context-switching,

system daemons, CPU frequency scaling, and garbage collection.

The temperature of the CPU being a factor is mitigated by monitoring the temperature of the CPU while doing experiments and only doing measurements when the temperature is similar across different experiments. CPU behavior differs across temperature ranges, as such experiments are to be performed within the same temperature range or established temperature intervals.

The OS context-switching randomly is mitigated by taking enough measurements to get a significant result, as explained in Section 4.5 and by shielding the cores used for benchmark execution to avoid uninvited threads as described in [102].

4.6.3 External Validity

External validity refers to whether the results from these experiments can be generalized.

There are several different external threats to validity, these include a generalization of language constructs to entire programs and the computer setup.

We consider the generalization of language constructs to an entire program to be a threat to validity, as larger programs may be compiled in a different way than individual experiments. This could give different energy efficiency results, compared to looking at the individual constructs.

The computer setup is a threat to validity as different computers may have different hardware optimizations that can have an effect on the energy efficiency of some programs, meaning the results could be different on another computer.

The generalization of language constructs to an entire program threat can be mitigated by testing if the results can be recreated in larger programs. The computer setup can be mitigated by doing the experiments on a representative suite of computers, however as computers constantly change and computer setups have a broad range of specifications, we do not test on multiple computers. Therefore we are conscious that the results we find may change were they performed on a different computer setup.

4.6.4 Reliability

Reliability refers to whether the results are reproducible if you use the same methodology, with the same setup and the same tests.

The biggest threat to reliability is the randomness of context-switching, which can be mitigated by shielding the cores used by the benchmark from other processes and by doing enough measurements to get a significant result. In addition there is a lot of randomness with hardware states that can have an impact on the results.

Preliminary results have shown that unrolling loops manually gives unreliable results compared to utilizing loops (seen in Section A.1.1), we hypothesize that this is because C# utilizes Just-In-Time (JIT) compilation. That being said, the unreliable results still occur when utilizing R2R compilation (seen in Section A.1.2), so it is unclear why that is the case. Besides this, preliminary testing has shown that the first result from benchmarks is a significant outlier. We hypothesize that it is because of JIT as during the first run, the code will be compiled during the run and afterwards the compiled code can be run without problems. However, more research into this is needed, which we explain in Chapter 10.

Furthermore, there are tips for how to reproduce results on Linux systems in [102]. These tips can be used to further mitigate reliability issues.

These tips include:

- shielding CPUs from uninvited threads, to make sure the CPU is not being used by something else while running an experiment,
- changing the swappiness parameter, so the Linux distribution does not swap as aggressively,
- turning off or creating a Address Space Layout Randomization (ASLR)-disabled shell to not create variations in memory layout,
- making the clock rate effectively static by using the "performance" governor on all CPUs and turning off boosting on all CPUs,
- disabling hyperthreading to make it easier to manage CPU resources, and
- sending interrupt requests to unshielded processors instead of the processors used for testing.

4.7 Summary

The overall goal of this study is to determine whether there is a difference in energy consumption between language constructs that achieve similar

outcomes in C#. Based on the goal of the study, a suite of benchmarks needs to be created and executed. Benchmarking presents a number of issues, as it is important for the benchmarks to be relevant, reproducible, fair, verifiable and usable, as presented in Section 4.1.

Furthermore, we look at two types of benchmarks, microbenchmarks and macrobenchmarks. We choose to look at microbenchmarks as we want to look at specific language constructs instead of larger programs, as presented in Section 4.2. At the time of writing, no suite of microbenchmarks that follow our requirements can be found, meaning we create our own microbenchmarks, as presented in Section 4.3. To measure these benchmarks, we use RAPL, as this has shown to give accurate measurements of power consumption, as presented in Section 4.4.

To make it possible to reproduce the results, we have set up guidelines on how to create and execute the microbenchmarks. We follow the approach recommended in [96] with a few differences, the main one being the number of times the benchmarks are run. We also describe how the code of the microbenchmarks should be set up, as presented in Section 4.5.

Lastly, we describe the different threats to validity that have an effect on the experiments. These are with regards to construct validity, internal validity, external validity and reliability. Here we describe the threats and how to mitigate them, for example by shielding the cores used by a benchmark to minimize the randomness of context-switching and doing validation of the results, as presented in Section 4.6.

Chapter 5

Experiment Execution

In this chapter we describe the setup of the experiments and their implementation. First we describe the setup of the experiments with the hardware specifications and efforts to minimize threats to validity. After this, we look at how we collect and run benchmarks. This is followed by how we do analysis on the results gathered from running the benchmarks. We give an example of how to use the benchmarks and perform analysis on them. The chapter ends with a summary of its contents. Important to note, not all of the code is described in this chapter, for example our implementation of Command-Line Interface (CLI) communication is not described. The full explanation of the code can be seen in Section A.2, and all of the code can be found on our GitLab repository [103].

5.1 Measurement Setup

For consistency we measure everything on a single computer with the specifications seen in Table 5.1.

Processor	Intel Xeon W-1250P @ 4.1 GHz
Storage	512 GB NVMe SSD
Memory	16 GB
Operating System	Ubuntu Server 20.04.03 LTS
.NET SDK	.NET 6.0.100

Table 5.1: Specifications for the computer used in testing.

Having one computer run all of the tests makes the results more com-

parable, as different computers do not consume the same amount of energy, use the same amount of time, etc. Beyond this, using a processor that builds upon the Comet Lake microarchitecture, which is based on the Skylake microarchitecture [104, 105], ensures that Running Average Power Limit (RAPL) is supported [27]. Besides this, we use `.NET 6.0.100` as the .NET SDK, which includes the compiler and makes it possible for us to use C# 10 features.

Furthermore, to improve comparability of the results, we follow the tips mentioned in Section 4.6.4.

Other than those tips, we run the tests in "Release" configuration as it creates less overhead when running the tests, than using the "Debug" configuration [106].

Writing Benchmarks

When writing the benchmarks, we have a three step process to mitigate errors in the benchmarks.

1. Write the benchmark/modify the benchmark,
2. let another project member review the benchmark, and
3. run the benchmark to see if the code is compiled away.

When checking if the code is compiled away, we compare the measurements of the benchmark written to the results of running an empty loop benchmark, this ensures that all the code inside the loop in the benchmark is not compiled away. We use this approach as we do not have the ability to look at the assembly code for the benchmarks, without using debug mode. Debug mode does not optimize code away making this the most reliable approach we have found.

If the benchmark is shown to be compiled away, we go through the process iteratively until we create a benchmark that is accepted and is not compiled away.

Lastly, we do not use the benchmarks present in [4] and [11] as:

- the benchmarks in those two papers get optimized away by the C# compiler,
- focus on too many language constructs or the benchmarks are created in a way that is not comparable between the benchmarks in the same group.

These issues are explored and shown in Section A.1.8.

5.2 Measurement Implementation

To run benchmarks and measure the runtime, package and DRAM energy, we build upon previous work done in [9]. Specifically, we build on their RAPL library for C#.

5.2.1 Running Benchmarks

To allow running multiple benchmarks automatically, we create two classes to setup the prerequisites for running benchmarks and collecting benchmarks. These classes are called `BenchmarkSuite` and `BenchmarkCollector` respectively.

```
1  public class BenchmarkSuite {
2      ...
3      public void RunAll() {
4          if (Environment.OSVersion.Platform != PlatformID.Unix) {
5              throw new NotSupportedException("Running the
                   ↪  benchmarks is only supported on Unix.");
6          }
7          List<IBenchmark> benchmarks =
           ↪  Benchmarks.OrderBy(benchmark =>
           ↪  benchmark.Order).ToList();
8          Warmup();
9          foreach ((int index, IBenchmark bench) in
           ↪  benchmarks.WithIndex()) {
10             bench.Run();
11         }
12         PlotGroups();
13         if (CsharpRAPLCLI.Options.ZipResults) {
14             ZipResults();
15         }
16     }
17     ...
18 }
```

Listing 1: Running all the benchmarks with the `RunAll` method.

In the `BenchmarkSuite` class, we create a method called `RunAll` which serves to run all the created benchmarks, seen in Listing 1. The `RunAll` method is created to allow running multiple benchmarks automatically, as the alternative is to run each individual benchmark by themselves. Here we call the `Warmup` method before we run the benchmarks, allowing the CPU to get ready thus minimizing outliers (as seen in Section A.1.6). After the warmup we run all of the benchmarks, after which we plot the results to boxplots, and at the end the results are zipped if specified by the user.

```
1  public class BenchmarkSuite {
2      ...
3      private static void Warmup() {
4          int warmup = 0;
5          for (int i = 0; i < 10; i++) {
6              while (warmup < int.MaxValue) {
7                  warmup++;
8                  ... // Write percentage completion to console
9              }
10             warmup = 0;
11         }
12     }
13     ...
14 }
```

Listing 2: Warming up to decrease number of outliers, with the `Warmup` method.

Creating a warmup method is necessary to minimize outliers, this warmup method serves to get the CPU running before the benchmarks are run. In Listing 2 the `Warmup` method can be seen, this method busy-waits for a while, specifically it counts an integer up until max value 10 times after which it has warmed up. The result of using a warmup method versus not using a warmup method can be seen in Section A.1.6.

```
1  public class BenchmarkCollector : BenchmarkSuite {
2      ...
```

```
3       private void CollectBenchmarks(Assembly assembly) {
4           foreach (MethodInfo benchmarkMethod in
            ↪  assembly.GetTypes().SelectMany(type =>
            ↪  type.GetMethods().Where(info =>
            ↪  info.GetCustomAttribute<BenchmarkAttribute>() !=
            ↪  null))) {
5               var benchmarkAttribute =
                ↪  benchmarkMethod.GetCustomAttribute<BenchmarkAttribute>()!;
6               if (benchmarkAttribute.Skip) {
7                   continue;
8               }
9               CheckMethodValidity(benchmarkMethod);
10              if (!_registeredBenchmarkVariations.
                ↪  ContainsKey(benchmarkMethod.ReturnType)) {
11                  RegisterAddBenchmarkVariation(benchmarkMethod);
12              }
13              (MethodInfo genericRegisterBenchmark, Type funcType)
                ↪  =
                ↪  _registeredBenchmarkVariations[benchmarkMethod.ReturnType];
14              Delegate benchmarkDelegate =
                ↪  benchmarkMethod.IsStatic ?
                ↪  benchmarkMethod.CreateDelegate(funcType) :
                ↪  benchmarkMethod.CreateDelegate(funcType,
                ↪  Activator.CreateInstance(benchmarkMethod.DeclaringType!));
15              genericRegisterBenchmark.Invoke(this, new object[]
                ↪  {benchmarkAttribute.Group!, benchmarkDelegate,
                ↪  benchmarkAttribute.Order});
16          }
17      }
18      ...
19  }
```

Listing 3: Collecting the benchmarks using reflection, with the `CollectBenchmarks` method.

Instead of sending a list with benchmarks to the `BenchmarkSuite`, we collect benchmarks automatically simplifying the process of running benchmarks for the developer. In Listing 3 we can see the `CollectBenchmarks` method. We utilize reflection in the `CollectBenchmarks` method to collect

every benchmark. Specifically, we look through the assembly (i.e. the files in the project) for every method that has the `Benchmark` attribute. We do this by looking through all the files and classes to find all the methods checking if they have a `BenchmarkAttribute` assigned to them. If they do, we check if the method is valid and map the return type to a method that can register a benchmark with this return type after which we get this method. We then create a delegate with this method so we can invoke it with the benchmark so we can add the benchmark to a list and call it later when we run all the benchmarks.

```
1 public class BenchmarkCollector : BenchmarkSuite {
2     ...
3     private void RegisterAddBenchmarkVariation(MethodInfo
      ↪  benchmark) {
4         Type funcType =
          ↪  typeof(Func<>).MakeGenericType(benchmark.ReturnType);
5         MethodInfo genericAddBenchmark =
          ↪  RegisterBenchmarkGenericMethod.MakeGenericMethod
          ↪  (benchmark.ReturnType);
6         _registeredBenchmarkVariations.Add(benchmark.ReturnType,
          ↪  new RegisterBenchmarkVariation(genericAddBenchmark,
          ↪  funcType));
7     }
8     ...
9 }
```

Listing 4: Registering `RegisterBenchmark` variations with different return types.

In Listing 4 we can see how we register a variation of the `RegisterBenchmark` method. The reason we need to do this, is because we cannot call generic methods with types gathered via reflection directly, as the return type of the methods gathered are not known at compile-time. Because of this, we create a variation of the `RegisterBenchmark` method with the correct return type with the use of the `MakeGenericMethod` method. We then add this variation to the `Dictionary` that contains the variations of `RegisterBenchmark` so it can be used later instead of creating a new variation for every method.

```
1 public class Benchmark<T> : IBenchmark {
2     ...
```

```
3       public void Run() {
4           Console.SetOut(_benchmarkOutputStream);
5           _rapl = new RAPL();
6           ElapsedTime = 0;
7           ...
8           if (CsharpRAPLCLI.Options.UseIterationCalculation) {
9               Iterations = IterationCalculationAll();
10          }
11          SetLoopIterations(10);
12          _normalizedReturnValue = _benchmark()?.ToString() ??
            ↪ string.Empty;
13          for (var i = 0; i <= Iterations; i++) {
14              ...
15              Start();
16              T benchmarkOutput = _benchmark();
17              End(benchmarkOutput);
18              ...
19          }
20          ...
21      }
22      ...
23  }
```

Listing 5: Running individual benchmarks using the `Run` method 1/3.

After creating a way to run all of the benchmarks and collecting all of the benchmarks automatically, we need to implement how to run each individual benchmark. This is seen in Listing 5. When running the benchmarks, we need to figure out how many times each benchmark must run to mitigate noise in the results. To do this, we call the `IterationCalculationAll` method, which uses the equation described in Equation 4.2 to calculate the number of iterations. Important to note is that we have a minimum number of 10 iterations, which allows gathering a sample before knowing if more iterations are needed. After the number of iterations is calculated, we run a normalized test to keep track of the return value. This is done because different benchmarks may run different amounts of times and may therefore produce different return values. Once this is done, we run the benchmarks by calling the `Start` method which sets up `RAPL`, followed by calling the actual benchmark that has been implemented and afterwards calling the `End`

method which gathers the results.

```
public class Benchmark<T> : IBenchmark {
    ...
    public void Run() {
        ...
        for (var i = 0; i <= Iterations; i++) {
            ...
            if (_loopIterationsFieldInfo != null &&
            ↪  CsharpRAPLCLI.Options.UseLoopIterationScaling &&
            ↪  _rawResults[^1].ElapsedTime <
            ↪  TargetLoopIterationTime) {
                int currentValue = GetLoopIterations();
                switch (currentValue) {
                    ...
                    default:
                        SetLoopIterations(currentValue +
                        ↪  currentValue);
                        _rawResults.Clear();
                        _normalizedResults.Clear();
                        i = 0;
                        continue;
                }
            }
            ...
        }
        ...
    }
}
```

Listing 6: Running individual benchmarks using the `Run` method 2/3.

After each iteration of the benchmark, we check if the benchmark has run for 0,25 seconds so we can gather results with less interference from other processes [96], as prescribed in Section 4.5. This is seen in Listing 6. If the benchmark has run for less than 0.25 seconds, we clear the results and double the amount of `LoopIterations` in the benchmark, effectively doubling the benchmark runtime.

```
public class Benchmark<T> : IBenchmark {
    ...
    public void Run() {
        ...
        for (var i = 0; i <= Iterations; i++) {
            ...
            if (ElapsedTime < MaxExecutionTime) {
                if (CsharpRAPLCLI.Options.
                ↪  UseIterationCalculation) {
                    Iterations = IterationCalculationAll();
                }
                continue;
            }
            break;
        }
        SaveResults();
        HasRun = true;
        ...
    }
}
```

Listing 7: Running individual benchmarks using the `Run` method 3/3.

After checking if we need to double the amount of `LoopIterations`, we recalculate the number of `Iterations` needed, as the number is dependent on the results gathered as seen in Listing 7. We do this by calling the `IterationCalculationAll` method. After recalculating the number of iterations we continue in the loop. Once the loop is done we save the results and set the variable `HasRun` to true.

```
public class Benchmark<T> : IBenchmark {
    ...
    private void End(T benchmarkOutput) {
        if (_rapl is null) {
            throw new NotSupportedException("Rapl has not been
            ↪  initialized");
        }
        _rapl.End();
```

```
8           if (!_rapl.IsValid()) {
9               return;
10          }
11          BenchmarkResult result = _rapl.GetResults() with {
12              BenchmarkReturnValue = benchmarkOutput?.ToString()
                ↪  ?? string.Empty
13          };
14
15          BenchmarkResult normalizedResult =
            ↪  _rapl.GetNormalizedResults(GetLoopIterations()) with
            ↪  {
16              BenchmarkReturnValue = _normalizedReturnValue ??
                ↪  string.Empty
17          };
18          _rawResults.Add(result);
19          _normalizedResults.Add(normalizedResult);
20          ElapsedTime += _rawResults[^1].ElapsedTime / 1_000;
21      }
22      ...
23  }
```

Listing 8: The `End` method in the `Benchmark` class.

In Listing 8 we can see how the results are gathered. We start by checking if `_rapl` is valid, because otherwise no results have been gathered. After checking, we get results in two different ways, the raw results used for calculating the amount of `Iterations` and the normalized results, which are used for comparing different benchmark results at a later stage. The results are normalized with regards to `LoopIterations`, which by default is 1.000.000, to make the results comparable.

5.2.2 Calculating Iterations

An important part of benchmarking is figuring out how many times to run a benchmark to get a representative result. To do this, we implement the calculation shown in Equation 4.2.

```
1   public class Benchmark<T> : IBenchmark {
2       ...
```

```
3       private int IterationCalculationAll(double confidence =
        ↪ 0.95) {
4           int dramIteration = IterationCalculation(confidence,
            ↪ BenchmarkResultType.DRAMEnergy);
5           int timeIteration = IterationCalculation(confidence,
            ↪ BenchmarkResultType.ElapsedTime);
6           int packageIteration = IterationCalculation(confidence);
7           return Math.Max(dramIteration, Math.Max(timeIteration,
            ↪ packageIteration));
8       }
9       ...
10  }
```

Listing 9: Calculating the Iteration for all measurement types.

In Listing 9 we can see that the method `IterationCalculation` is called for three types of `BenchmarkResultType`, specifically DRAM energy, elapsed time and Package energy. This is done to ensure that all three results are significant, in case the user wants to test for a specific one of them. The max value of the three types of `BenchmarkResultType` is returned.

```
1   public class Benchmark<T> : IBenchmark {
2       ...
3       private int IterationCalculation(double confidence = 0.95,
        ↪ BenchmarkResultType resultType =
        ↪ BenchmarkResultType.PackageEnergy) {
4           if (_rawResults.Count < 10) {
5               return 10;
6           }
7           double alpha = 1 - confidence;
8           var values = new double[_rawResults.Count];
9           for (var i = 0; i < _rawResults.Count; i++) {
10              values[i] = resultType switch {
11                  BenchmarkResultType.Temperature =>
                    ↪ _rawResults[i].Temperature,
12                  ...
13              };
14          }
```

```
15          double mean = values.Average();
16          double stdDeviation = values.StandardDeviation(mean);
17          NormalDistribution nd = new(mean, stdDeviation);
18          return (int) Math.Ceiling(Math.Pow(nd.ZScore(
            ↪  nd.GetRange(alpha / 2).Max) * stdDeviation / (0.005
            ↪  * mean), 2));
19      }
20      ...
21  }
```

Listing 10: Calculating the Iteration for a specific measurement type.

In Listing 10 we can see the implementation of Equation 4.2. Here we utilize the library Accord [107] to calculate the standard deviation and create a normal distribution.

We have 10 as the minimum number of measurements, as recommended by [96]. Once 10 measurements have been completed, we have a sample from which we can calculate the number of iterations needed for a significant result.

We start by calculating the α value by subtracting the confidence wanted from 1. After this, we create an array of doubles that has the same size as the number of results gathered so far. We add the relevant measures gathered to the values array, after which we calculate the mean and standard deviation of these values. With the mean and standard deviation, we can create a normal distribution which we use to calculate the z-score for the equation.

At the end, we put in the numbers as seen in Equation 4.2 and return the calculated amount of iterations necessary. We calculate the z-score by giving the `ZScore` method the highest value in the first $alpha/2$ part of the normal-distribution, as the `ZScore` method takes a value instead of a percentage.

5.3 Analysis Implementation

Once the benchmarks have been run and the results have been measured, we want to analyse the results. Specifically, we want to know if there are significant differences (i.e. a p-value below 0,05) between each benchmark.

```
1   public class Analysis {
2       ...
```

```
3       public static Dictionary<string, double>
        ↪ CalculatePValueForGroup(List<IBenchmark> dataSets) {
4           var groupToPValue = new Dictionary<string, double>();
5           for (int i = 0; i < dataSets.Count; i++) {
6               for (int j = i + 1; j < dataSets.Count; j++) {
7                   var analysis = new Analysis(dataSets[i],
                    ↪ dataSets[j]);
8                   foreach ((string message, double value) in
                    ↪ analysis.CalculatePValue()) {
9                       groupToPValue.Add(message, value);
10                  }
11              }
12          }
13          return groupToPValue;
14      }
15  }
```

Listing 11: Calculating the *p*-value for a group of results in the `Analysis` class.

In Listing 11 we can see how we calculate the *p*-values for all benchmarks in a group of benchmarks. We do this by going through all the results, creating `Analysis` objects for each pair of these benchmarks and calling the `CalculatePValue` method on these, after which we add the result to the `Dictionary` returned from this method.

```
1   public class Analysis {
2       ...
3       public List<(string Message, double Value)>
        ↪ CalculatePValue() {
4           PValueData firstDataSet = new(_firstDataset);
5           PValueData secondDataSet = new(_secondDataset);
6           var timeTTest = new
            ↪ TwoSampleTTest(firstDataSet.TimesValues,
            ↪ secondDataSet.TimesValues);
7           var pkgTTest = new
            ↪ TwoSampleTTest(firstDataSet.PackageValues,
            ↪ secondDataSet.PackageValues);
```

```
8           var dramTTest = new
            ↪  TwoSampleTTest(firstDataSet.DRAMValues,
            ↪  secondDataSet.DRAMValues);
9           return new List<(string Message, double Value)> {
10              ($"{firstDataSet.Name} significantly different from
                ↪  {secondDataSet.Name} - Time", timeTTest.PValue),
11              ($"{firstDataSet.Name} significantly different from
                ↪  {secondDataSet.Name} - Package",
                ↪  pkgTTest.PValue),
12              ($"{firstDataSet.Name} significantly different from
                ↪  {secondDataSet.Name} - DRAM", dramTTest.PValue)
13          };
14      }
15      ...
16  }
```

Listing 12: Calculating the *p*-value between two datasets in the `Analysis` class.

In Listing 12 we can see how we calculate the *p*-value for a pair of results. We use [107] to help with the calculation of the *p*-value. We create two `PValueData` objects with the results that were used when creating the `Analysis` object. After this, we create a `TwoSampleTTest` object for each of the result types and at the end return the *p*-value with a message describing what the *p*-value shows.

5.4 Default Implementation

Now that all of the code for creating, running, and analysing benchmarks have been implemented, we create a default implementation that shows how to run and analyse the benchmarks.

```
1   CsharpRAPLCLI.SetAnalysisCallback(_ => {
2       // Do analysis
3   });
4   Options options = CsharpRAPLCLI.Parse(args);
5   if (options.ShouldExit) {
6       return;
```

```
}
var suite = new BenchmarkCollector(options.Iterations,
    options.LoopIterations);
```

Listing 13: Getting ready to run and analyse the benchmarks.

In Listing 13 we can see an example of how to setup everything needed to run the benchmarks. We start by setting the analysis callback to an anonymous function, where we can implement our own analysis. If no callback is implemented, the only post-processing done is creating plots and *csv* files containing the measurements. After this, we set up an `Options` object, with the help of the `Parse` method on `CsharpRAPLCLI`, so any options communicated via the CLI are updated. We then check if we should exit, in which case we just return, and thereby exit. For example, if we want to only analyse earlier results. Lastly, we set up a `BenchmarkCollector` object, which we use to run all of the benchmarks.

```
suite.RunAll();
foreach ((string group, List<IBenchmark> benchmarks) in
    suite.GetBenchmarksByGroup()) {
    Dictionary<string, double> result =
        Analysis.CalculatePValueForGroup(benchmarks);
    DateTime dateTime = DateTime.Now;
    var time = $"{dateTime.ToString("s").Replace(":",
        "-")}-{dateTime.Millisecond}";
    Directory.CreateDirectory(Path.Join(options.OutputPath,
        $"_pvalues/{group}/"));
    using var writer = new
        StreamWriter(Path.Join(options.OutputPath,
        $"_pvalues/{group}/{time}.csv"));
    using var csv = new CsvWriter(writer, new
        CsvConfiguration(CultureInfo.InvariantCulture) {
        Delimiter = ";" });
    csv.WriteRecords(result);
}
```

Listing 14: Running and calculating the *p*-value of all benchmarks.

In Listing 14 we can see how the benchmarks are run and how we get the *p*-values between all the benchmarks. At the start we can see all the benchmarks are run by calling the `RunAll` method on the `suite` object. After this we go through all of the benchmarks by group, on which we call the `CalculatePValueForGroup` method from the `Analysis` class. After this we create a *csv* file containing all the *p*-values in the output path.

```
1 using var zipToOpen = new FileStream("results.zip",
  ↪ FileMode.Open);
2 using var archive = new ZipArchive(zipToOpen,
  ↪ ZipArchiveMode.Update);
3 foreach (string file in
  ↪ Directory.EnumerateFiles(Path.Join(options.OutputPath,
  ↪ "_pvalues/"), "*.csv",
4     SearchOption.AllDirectories)) {
5     archive.CreateEntryFromFile(file, file);
6 }
7 archive.Dispose();
8 zipToOpen.Dispose();
9 CsharpRAPLCLI.StartAnalysis(suite.GetBenchmarksByGroup());
```

Listing 15: Saving the *p*-values to the zip file containing the results and calling the analysis callback.

In Listing 15 we can see how the *p*-values are saved to the zip archive containing the results. We then call the `StartAnalysis` method in the `CsharpRAPLCLI` class, which calls the analysis callback.

5.5 Summary

We use a computer with the specifications seen in Table 5.1 to run the benchmarks. This computer has a processor that builds upon the Comet Lake microarchitecture [108], making it compatible with RAPL.

We have created a framework that automatically gathers all the created benchmarks, and skips the ones that the developer wants to skip. This allows a developer to implement benchmarks and immediately run them without needing extra setup. The code that collects the benchmarks also

automatically runs the benchmarks allowing the developer to call a single method and letting the program do the rest. We created automatic warmup for the benchmarks, so the developer does not need to worry about this aspect of benchmarking code. Furthermore, the developer does not need to set a specific amount of iterations for a benchmark, as this is calculated automatically. The saved results are automatically normalized with regards to `LoopIterations`, so a developer can immediately analyse the results.

After running the benchmarks, analysis can be done with the help of built-in functionality, for example seeing if there is a significant difference for the elapsed-time, package energy and DRAM energy between the benchmarks by calculating the p-value.

Lastly, we create a default implementation that shows how to utilize the functionality within the program.

Chapter 6

Results

In this chapter we look at the results we have obtained from our experiments. All the benchmarks that have been used to reach these results can be found in our git repository [103]. Important to note is that all results have been normalized to 1.000.000, as it makes it possible to compare different benchmarks.

The results we show in this chapter are chosen because we consider them unexpected, for a look at all of the results see Section A.3. In Chapter 7 we analyse the results shown in this chapter.

Procedure

The procedure for each section is:

1. Give a short description of the benchmarks, what are the constructs used and when are they applicable.
2. Show code for one of the benchmarks.
3. Show one or more plots of the results, to give a quick overview of the results. In the plots, the dashed lines show the average, the solid lines show median, the boxes show the 95% interval within the results found, and the outer lines show the furthest outliers. In some cases the average and median is on top of each other, and therefore hard to see. The outliers in some cases have happened so often they are within the 95%.
4. Show the table with the numeric results to give a more detailed insight into the results.

6.1 Primitive Integer

In this section, we look at the efficiency of the different primitive integer datatypes. We have created benchmarks for the following primitive integer datatypes:

- `sbyte`(Signed 8-bit)
- `short`(Signed 16-bit)
- `int`(Signed 32-bit)
- `long`(Signed 64-bit)
- `nint`(Architecture dependent, signed 32 or 64-bit)
- `byte`(Unsigned *byte*)
- `ushort`(Unsigned `short`)
- `uint`(Unsigned `int`)
- `ulong`(Unsigned `long`)
- `nuint`(Unsigned `nint`)

Each of these integer datatypes have varying sizes meaning that they can represent different ranges of numbers. Unsigned datatypes can only represent positive numbers and since they do not represent the negative numbers their range of positive numbers is increased.

We start by looking at the benchmark `Int`.

```
1  public class PrimitivesBenchmarks {
2      ...
3      [Benchmark("PrimitiveInteger", "Tests operation on primitive
   ↪  int")]
4      public static int Int() {
5          int primitive = 0;
6          for (int i = 0; i < LoopIterations; i++) {
7              primitive++;
8              primitive *= 3;
9              primitive /= 2;
10             primitive--;
11             primitive %= 20;
12         }
13         return primitive;
14     }
15     ...
16 }
```

Listing 16: The `Int` method which tests operations on `int` type in the `PrimitiveBenchmarks` class.

To begin we look at how we have created the benchmark for each of the datatypes. In Listing 16 we can see how the `int` datatype benchmark is created. We do five different operations on the `primitive` variable, for as many times as the variable `LoopIterations` has been set to, after which the `primitive` variable is returned. The same is the case for all the other primitive integer datatype benchmarks[103].

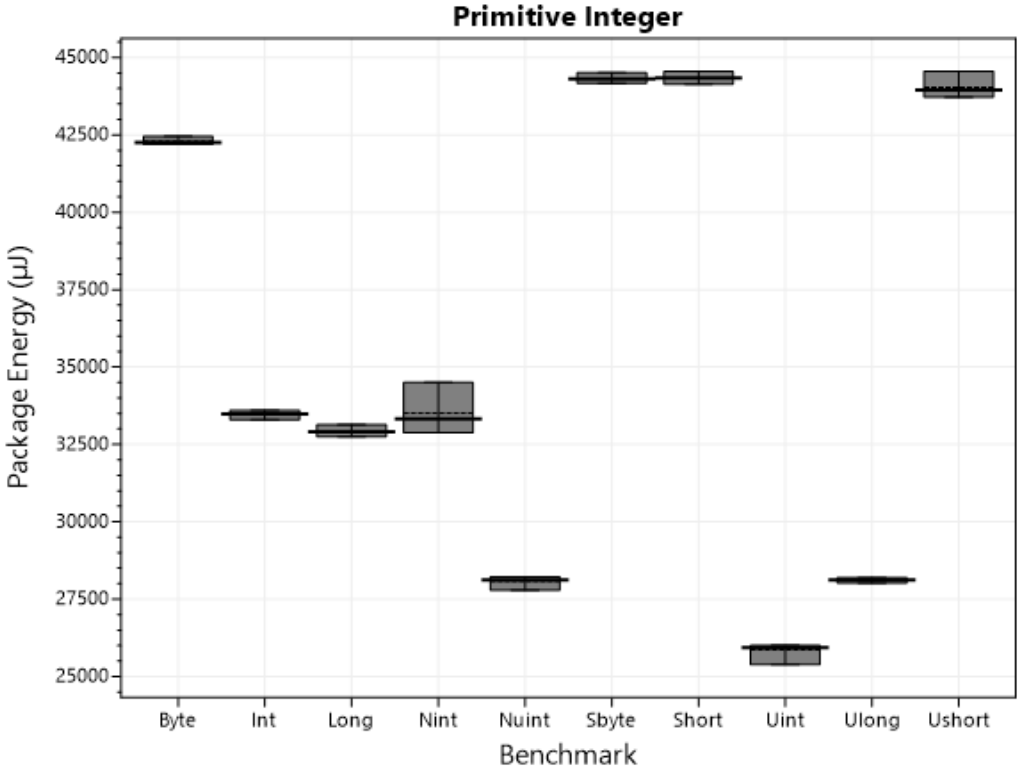

Figure 6.1: Boxplot showing how efficient different Primitive Integer types are with regards to Package Energy.

In Figure 6.1 we can see differences existing between primitive integer data types. The y-axis does **not** start from zero to show a better view of the differences, and the span is around 20.000 µJ.

Here we can see the most efficient datatype with regards to Package Energy consumption is the `uint` datatype, while the least efficient is close

to a tie between the `short` and the `sbyte` datatypes.

We can also see that unsigned integers are at least 1% more efficient than their signed variant, except for the case of `ushort` and `short`.

With regards to signed integers, we can see that the most efficient are `nint` (native integer), `long` and `int` with no large difference between the three.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Byte	4,774144	42.291,558	2.654,990
Int	3,770063	33.451,857	2.100,078
Long	3,723723	32.924,190	2.071,137
Nint	3,723477	33.503,079	2.071,678
Nuint	3,184053	28.078,657	1.769,831
Sbyte	5,014215	44.328,415	2.788,301
Short	5,015832	44.327,446	2.790,484
Uint	2,935853	25.862,204	1.633,408
Ulong	3,183120	28.112,855	1.768,374
Ushort	5,040331	44.033,497	2.804,712

Table 6.1: Table showing the elapsed time and energy measurement for each Primitive Integer.

In Table 6.1 we can see the results in numeric form. Specifically, we can see the average result of all three result types (Time elapsed, Package Energy consumed, and DRAM Energy consumed).

In Table 6.1 we can see that `ushort` is around 1% more efficient than the signed variant with regards to Package Energy, which is surprising given that all other unsigned datatypes are at least 1% more efficient compared to the signed counterpart.

6.1.1 Findings

These findings can then help determine what construct should be utilized. I.e. if as a developer you only need to represent positive numbers up to 4.294.967.295, `uint` would be the ideal choice if concerned with time, energy, or a combination of these. If you know that you need both negative and positive integers the result shows that `long` would be the best in regards to energy consumption. To summarize, the interesting findings are as follows:

- Unsigned integer datatypes are at least 1% more efficient than signed integer datatypes, except
- `ushort` is only 1% more efficient than `short`, and
- `uint` is the most efficient datatype.

6.2 Selection

In this section, we look at the efficiency of different selection statements. The section is split into four parts, `switch` subgroup comparison, `const` subgroup comparison, `if` subgroup comparison and `conditional` subgroup comparison. The four groupings are made because not all benchmarks in selection are comparable, but the benchmarks within the groups are.

6.2.1 Switch

In this section, we look at the efficiency of using switch statements. We have created two benchmarks in this group. These use the language constructs `switch` and `if`. The `switch` benchmark is constructed so it reflects the conventional use of `switch`. We see in Listing 17 how we have imitated a `switch` through the use of `if`.

```
...
    [Benchmark("SelectionSwitch", "Tests if statement compared
↪   to switch")]
    public static int IfComparableWithSwitch() {
        int count = 1;
        for (int i = 0; i < LoopIterations; i++) {
            if (count == 1) {
                count = 2;
                continue;
            }

            if (count == 2) {
                count = 3;
                continue;
            }
```

```
15  ...
16              if (count == 9) {
17                  count = 10;
18                  continue;
19              }
20
21              count = 1;
22          }
23
24          return count;
25      }
26  ...
```

Listing 17: The `IfComparableWithSwitch` method which tests `if` statement in the `SelectionBenchmarks` class.

In Listing 17 we show the `IfComparableWithSwitch` benchmark created for testing selection. We use a `for` loop to iterate over a sequence of nine `if` statements for `LoopIterations` times. This is done to make it comparable with a method using switch on nine cases, where the same general structure is used.

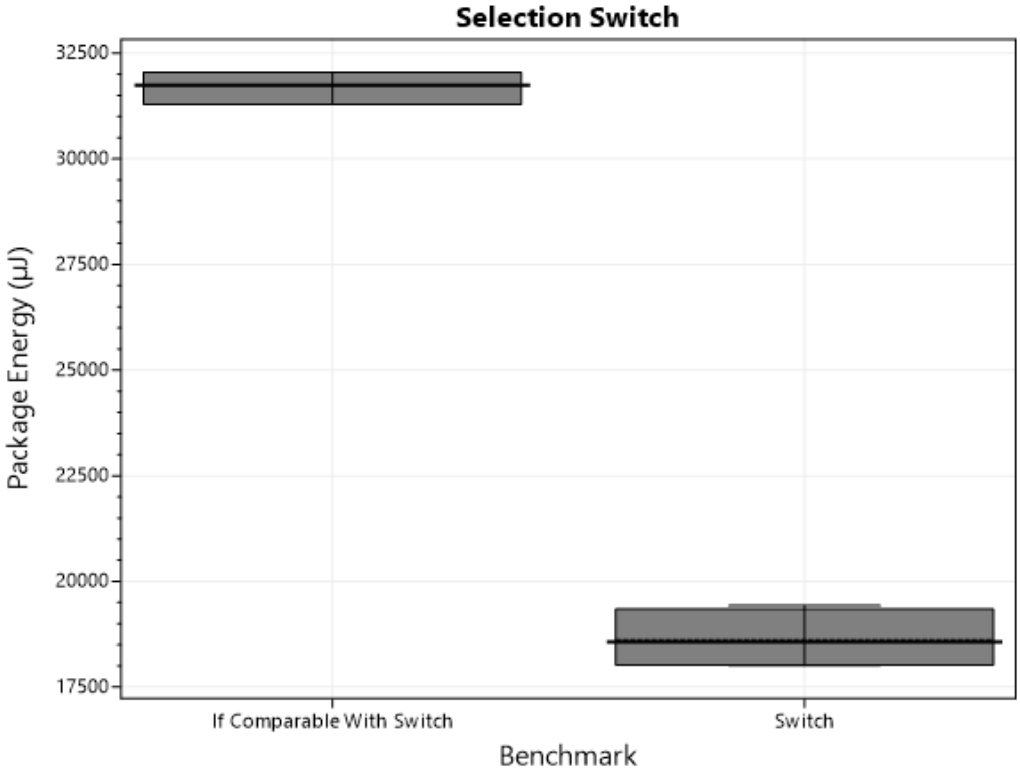

Figure 6.2: Boxplot showing how efficient different selection `switch` benchmarks are with regards to Package Energy.

In Figure 6.2 we can see that there are differences in package energy consumption between the selection switch benchmarks. The y-axis does **not** start from zero and spans around 15.000 µJ. We can see the `Switch` benchmark uses less energy than the `If Comparable With Switch`.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
If Comparable With Switch	2,973944	31.751,188	1.653,296
Switch	1,618432	18.620,357	900,318

Table 6.2: Table showing the elapsed time and energy measurement for different Selection benchmarks.

In Table 6.2 we can see the average results for our three result types, Time elapsed, Package energy consumed and DRAM Energy consumed. We can see that `Switch` is almost twice as fast as the `If Comparable With Switch`, and consumes less than 2/3 of the energy amount of `If Comparable With Switch`.

To see if the results are significant, we look at the p-values in the `switch` section of Section A.3. From this, we can see that there is a significant difference between `Switch` and `If Comparable With Switch`.

6.2.2 Const

In this section, we look at the efficiency of using constants in the condition of selection statements. This is done to be able to compare with switch statements which need to use constants. We make use of the constructs `switch` and `if`, in Listing 18 we show how `switch` is imitated through the use of `if`.

Looking at the benchmark for `IfConstNumber` also serves as an example of our benchmarks in our const group.

```
...
    public static int IfConstNumber() {
        const int halfLoopIteration = 25000;
        int count = 0;
        for (int i = 0; i < LoopIterations; i++) {
            if (i < halfLoopIteration) {
                count++;
                continue;
            }

            if (i == halfLoopIteration) {
                count += 10;
                continue;
            }

            if (i > halfLoopIteration) {
                count--;
                continue;
            }

            halfLoopIteration++;
        }

        return count;
```

```
25      }
26  ...
```

Listing 18: The `IfConstNumber` method in the `SelectionBenchmarks` class, which tests if statement using const to be comparable with a switch.

In Listing 18 we show the `IfConstNumber` benchmark created for testing selection. We use a `for` loop to iterate over three `if` statements `LoopIterations` number of times. We use a constant to make it comparable with `Switch` as `Switch` needs to use constants. The other benchmark using `Switch` uses the same general structure.

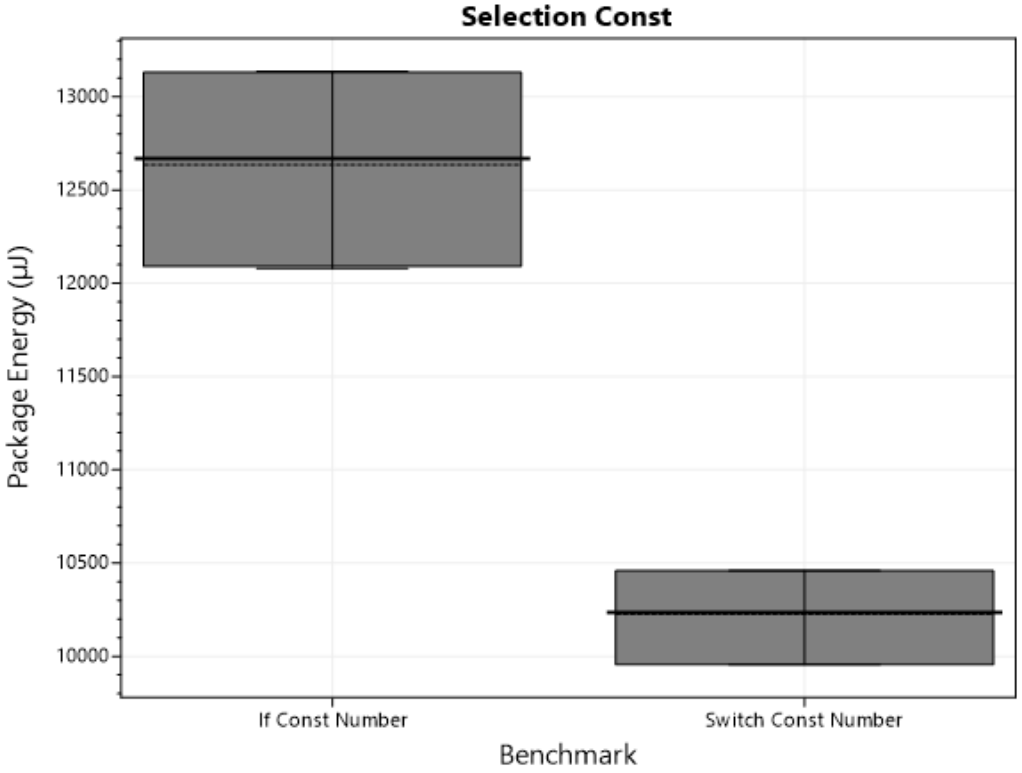

Figure 6.3: Boxplot showing how efficient different selection `const` benchmarks are with regards to Package Energy.

In Figure 6.3 we can see the difference in package energy for the two benchmarks in our const group. The y-axis does not start from zero and spans around 3.000 µJ. We see in the figure that the `Switch Const Number` benchmark consumes less energy than the `If Const Number` benchmark.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
If Const Number	1,163601	12.636,177	647,165
Switch Const Number	0,909927	10.226,356	506,081

Table 6.3: Table showing the elapsed time and energy measurement for each Selection Const.

In Table 6.3 we can see the average results for our three result types. We can see that `Switch Const Number` is faster than `If Const Number`, and consuming less energy.

To see if the results are significant, we look at the p-values in the `const` section in Section A.3. From this we can see that there is a significant difference with a p-value under 0,05 between `Switch Const Number` and `If Const Number`.

6.2.3 Conditional

In this section we look at the efficiency of using conditional selection.

We have constructed two benchmarks to look at this. We look at the conditional operator and `if-else`, to see if the use of one is preferable to another, is cases where both can be utilized. In Listing 19 we can see how the conditional operator is used, this is then compared to the equivalent using `if` and `else`.

```
1  ...
2      [Benchmark("SelectionConditional", "Tests if conditional
   ↪  operator")]
3      public static int ConditionalOperator() {
4          int count = 0;
5          for (int i = 0; i < LoopIterations; i++) {
6              count = i <= LoopIterations ? 1 : 2;
7          }
8
9          return count;
10     }
11 ...
```

Listing 19: The `ConditionalOperator` method which tests conditional operator in the `SelectionBenchmarks` class.

In Listing 19 we show the `ConditionalOperator` benchmark created for testing selection. We use a `for` loop to iterate for `LoopIterations` number of times. Inside the loop we set count to either 1 or 2 dependent on if `i` is smaller or equal to `LoopIterations`. We compare this benchmark with a benchmark which has the same structure, but uses an `if else` instead of the conditional operator.

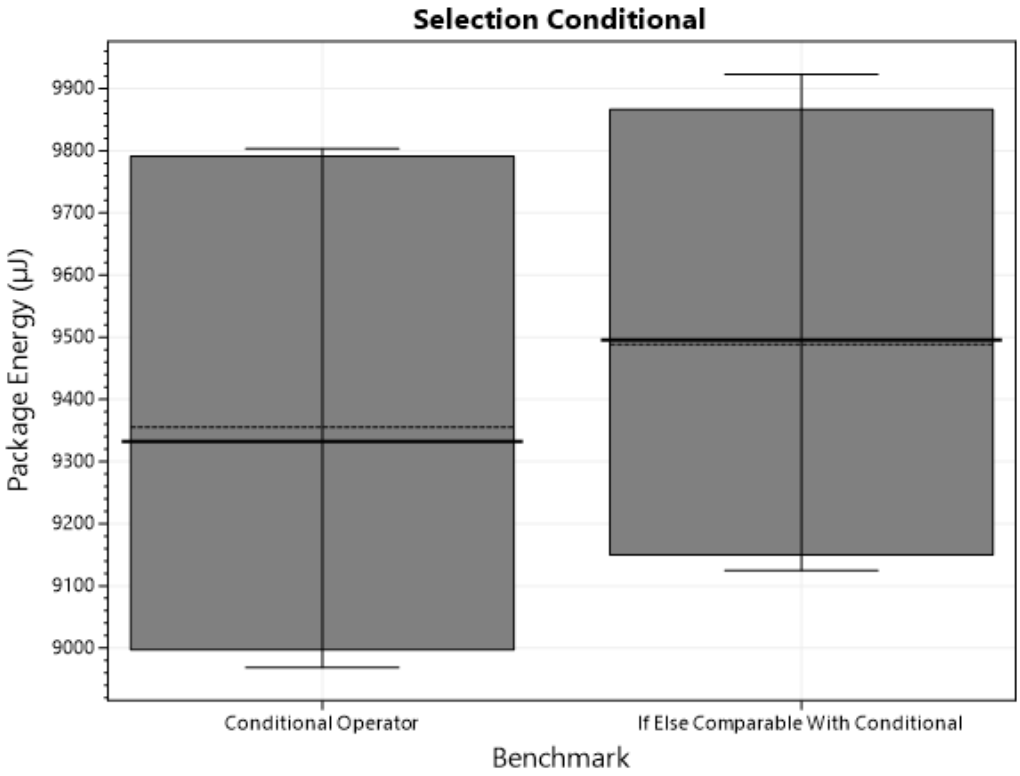

Figure 6.4: Boxplot showing how efficient different selection `conditional` benchmarks are with regards to Package Energy.

In Figure 6.4 we can see the difference in package energy consumption between the two benchmarks. The y-axis does not start from zero and spans around 1.000 µJ. We see that the benchmark using the `conditional operator` consumes the least energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Conditional Operator	0,897267	9.355,048	498,984
If Else Comparable With Conditional	0,896432	9.488,077	498,599

Table 6.4: Table showing the elapsed time and energy measurement for each Selection Conditional.

In Table 6.4 we see the average results for time, package energy consumed and DRAM energy consumed. We can see that `Conditional Operator` and `If Else Comparable With Conditional` almost have the same average run-time and DRAM energy consumption, however, the average package energy consumption is lower for `Conditional Operator`.

To see if the results are significant, we look at the p-values in the `conditional` section in Section A.3. From this we can see that there is a significant difference between the `Conditional Operator` benchmark and `If Else Comparable With Conditional` benchmark.

6.2.4 If Statements

In this section, we look at the efficiency of using if statements. This has been done using three benchmarks. One benchmark using purely `if`, another `if` and `else` and the last benchmark using `if`, `else if` and `else`. In Listing 20 we see how the benchmark using `if` and `else` has been constructed. Equivalent benchmarks are also constructed for the two others. This allows for determining of which combination of construct performs the best in regards to energy consumption.

```
...
    public static int IfElse() {
        int count = 0;
        int halfLoopIteration = LoopIterations / 2;
        for (int i = 0; i < LoopIterations; i++) {
            if (i < halfLoopIteration) {
                count++;
                continue;
            }

            if (i == halfLoopIteration) {
                count += 10;
                continue;
```

```
14              }
15              else {
16                  count--;
17                  continue;
18              }
19          }
20
21          return count;
22      }
23  ...
```

Listing 20: The `IfElse` method which tests if else statement in the `SelectionBenchmarks` class.

In Listing 20 we show the `IfElse` benchmark created for testing selection. We use a `for` loop to iterate over two `if` and one `else` statement for `LoopIterations` number of times. We have made three cases to compare them to an `if else if` statement. The other benchmarks comparable with `IfElse` have the same general structure.

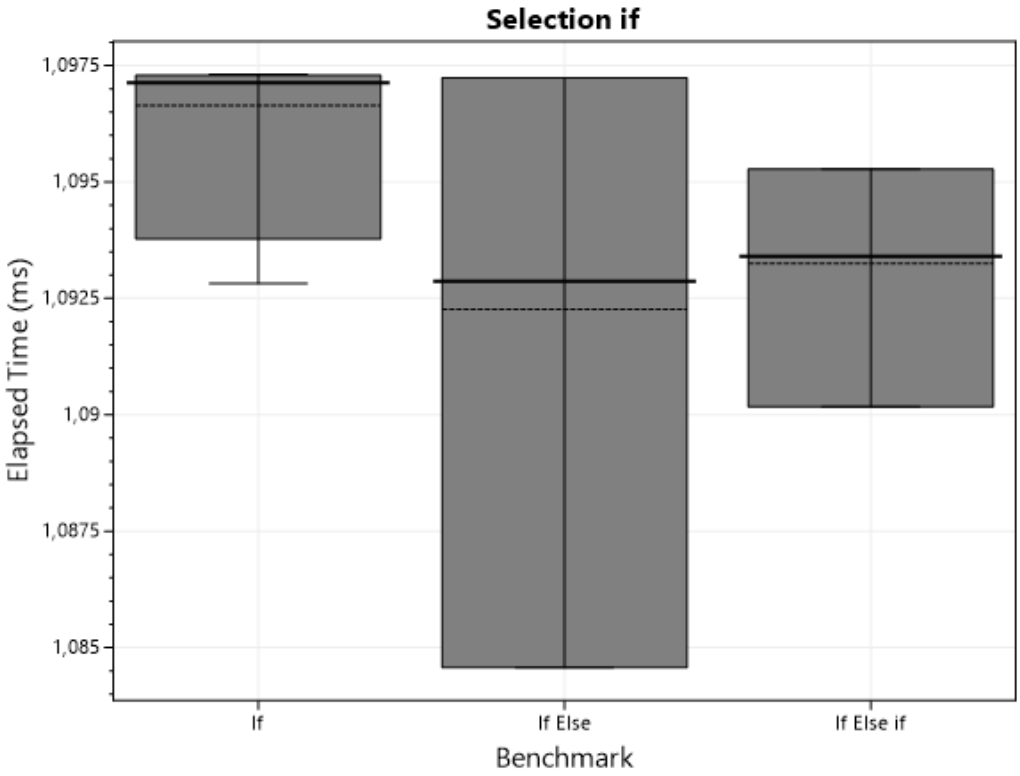

Figure 6.5: Boxplot showing how efficient different Selection if types are with regards to Elapsed Time.

In Figure 6.5 we see the difference in time elapsed, for the three benchmarks in our if selection grouping. The y-axis does not start from zero, and only spans around 0,015 ms. We see that the `If` benchmark has the highest elapsed time of the three.

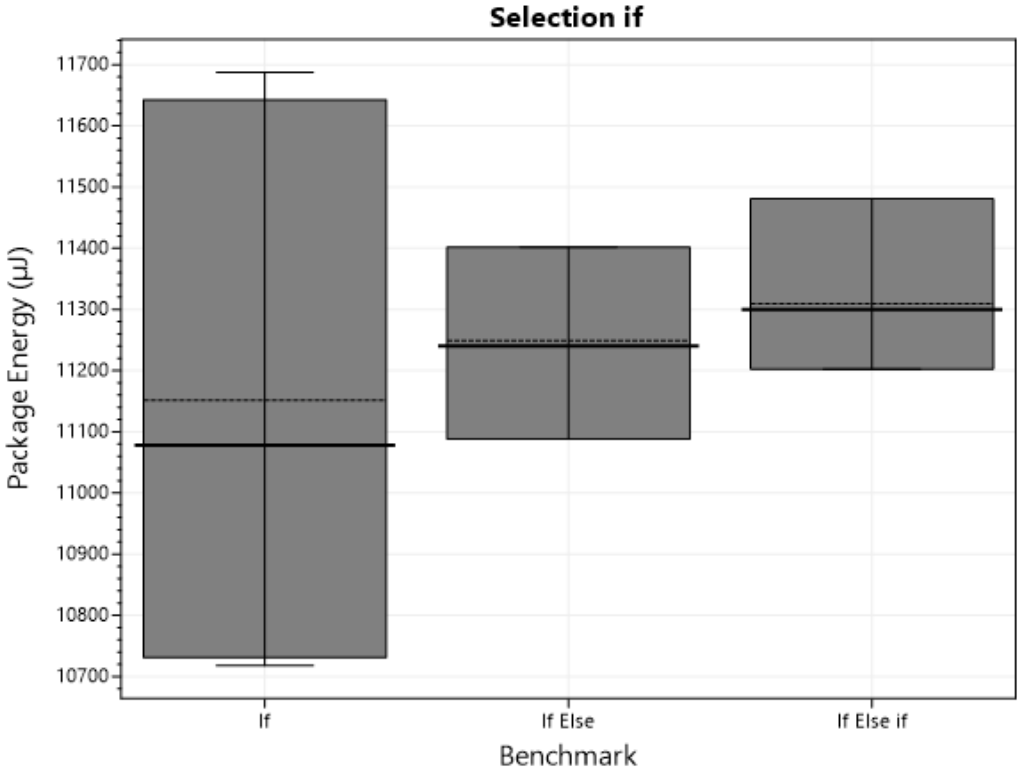

Figure 6.6: Boxplot showing how efficient different Selection if types are with regards to Package Energy.

In Figure 6.6 we see the difference in package energy consumption, for the three benchmarks in our `If` selection grouping. The y-axis does not start from zero, and spans around 1.000 µJ. We see that the `If` benchmarks consumes the least energy of the three benchmarks, while it has the highest elapsed time. This is interesting as time and energy consumption in most of our other results correlate.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
If	1,096640	11.151,124	609,902
If Else	1,092265	11.248,654	607,645
If Else if	1,093252	11.308,931	608,107

Table 6.5: Table showing the elapsed time and energy measurement for each Selection if.

In Table 6.5 we see the average results for time, package energy consumed and DRAM energy consumed. As we observed in the box plot `If`

has the highest time elapsed, however it is not a big difference, and the lowest package energy consumption. `If` has the highest DRAM energy consumption but still has the overall lowest energy consumption of the three benchmarks.

To see if the results are significant, we look at the `if statement` section in Section A.3, as it shows the *p*-value for elapsed time between the three benchmarks. We see that the `If` benchmark is significantly different in elapsed time from the other two benchmarks.

In the `if statement` section in Section A.3, we see the *p*-values for package energy consumption. We see that the difference between `if` and `if Else` is not significant enough to conclude that they consume different amounts of package energy, while we can conclude that `if` and `if else if` does consume a different amount of package energy.

6.2.5 Findings

The findings we have seen throughout this section can help a developer determine which language construct should be used in a given situation. We found that given the choice between performing selection using `switch` and `if`, `switch` is to be preferred across both time and energy.

Looking at these constructs once more, but using a constant number for selection, we found that once more `switch` is preferable to `if`.

The third group of benchmarks concerned themselves with what was preferable; the conditional operator or using `if` and `else`. We found that on average the conditional operator was better across the different metrics and thus preferable. However, in Figure 6.4 we saw that both constructs varied a great deal and as a developer, you might run into cases the conditional operator performs worse than the equivalent use of `if` and `else`.

In the last group of benchmarks regarding selection, we looked at which use of `if` is preferable. We can see that the choice here is dependent on what is the main concern for the developer the `if` concern is purely regarding energy consumption, results show that using `if`, is not significantly better than `if` and `else`, but it is better than `if, else if`. Given these results it is up to the developer to determine what they wish to use.

To summarize, the interesting findings in Selection are:

- `switch` is more efficient than `if` statements when applicable,
- conditional operators are more efficient than `if` statements when looking at package energy, and

- `if` consumes the least amount of energy while being the slowest of the benchmarks it is compared to.

6.3 Collections

In this section, we look at the efficiency of different collections. The section is split into three parts, first getting items from lists, removing items from lists, and getting items from tables. These groupings are made as each subcategory contains surprising results.

6.3.1 List Get

In this section, we look at the efficiency of different methods of accessing integers from different list types.

We have created benchmarks for five different list representations, which are:

- `array`
- `ArrayList`
- `ImmutableArray`
- `ImmutableList`
- `List`

In Listing 21 we can see how the benchmarks are built. The only difference between them is what list type is used, and what its method for retrieving an element is.

```
1 [Benchmark("ListGet", "Tests getting values sequentially from a
  ↪ ImmutableList")]
2 public static int ImmutableListGet() {
3     int sum = 0;
4     for (int i = 0; i < LoopIterations; i++) {
5         for (int j = 0; j < Data.Count; j++) {
6             sum +=
               ↪ Data[CollectionsHelpers.SequentialIndices[j]];
7         }
8     }
9
```

```
10      return sum;
11  }
```

Listing 21: The `ImmutableListGet` method which tests accessing data in an `Immutable List` in the `ImmutableListBenchmarks` class.

In Listing 21 we see the `ImmutableListGet` benchmark to test accessing elements in the `ImmutableList` named `Data` containing 1000 integers. We compare this benchmark with other benchmarks that test accessing integers from different types of lists, as well as accessing the elements in a well defined "random" order. These other benchmarks can be seen on our git repository[103].

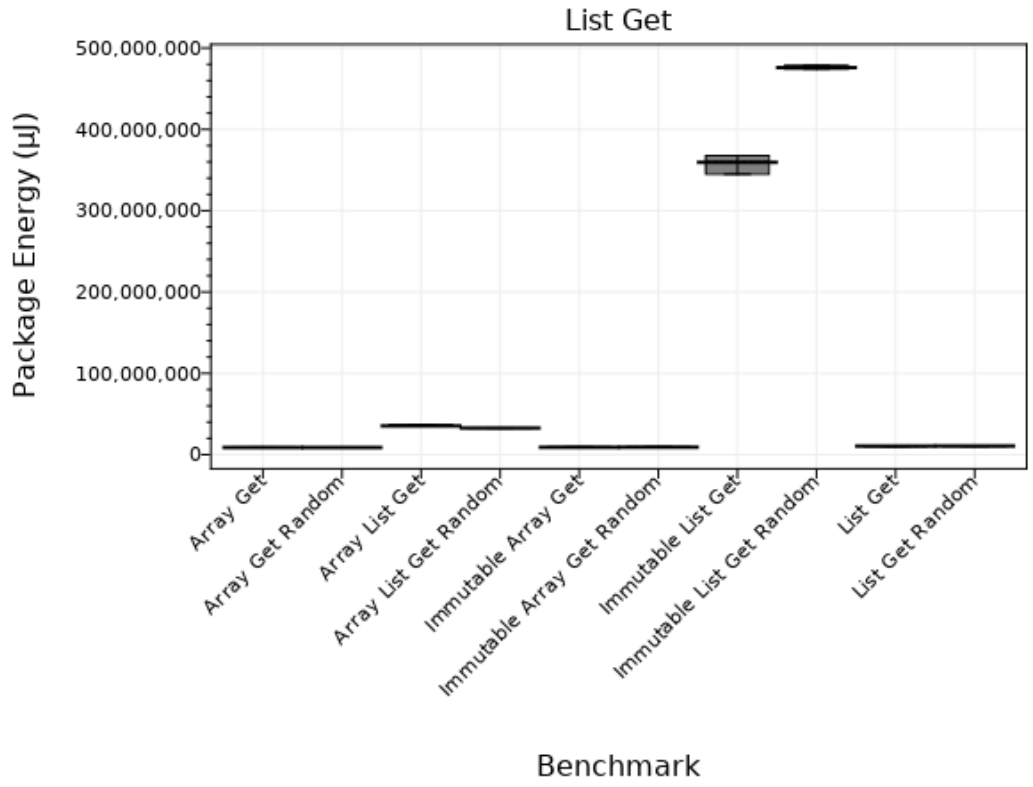

Figure 6.7: Boxplot showing how efficient different List Get methods are with regards to Package Energy.

In Figure 6.7 we see the package energy consumption for our benchmarks in the List Get group. The y-axis start from zero and spans around

500,000,000 µJ. In Figure 6.7 we see two significant outliers, those being `Immutable List Get` and `Immutable List Get Random`.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Array Get	617,467346	8.794.373,745	343.657,260
Array Get Random	617,463871	8.744.610,426	343.377,686
Array List Get	2.306,950468	35.512.334,487	1.283.994,428
Array List Get Random	2.140,269572	32.721.541,341	1.190.599,908
Immutable Array Get	638,782772	9.042.632,718	355.391,778
Immutable Array Get Random	638,834178	9.154.411,825	355.370,924
Immutable List Get	28.293,397352	359.966.427,951	15.739.062,500
Immutable List Get Random	39.624,610460	476.131.944,444	22.045.713,976
List Get	741,488156	10.531.627,401	412.543,742
List Get Random	741,498125	10.633.192,952	412.429,979

Table 6.6: Table showing the elapsed time and energy measurement for each List Get.

In Table 6.6 we see the numeric results, and that `Immutable List Get` uses about 34 times mores energy than `List Get`, and also almost 40 times more energy than the `Immutable Array get`. This is very surprising that specifically the `Immutable List` is way less efficient than it's non-immutable counterpart, but also the `Immutable Array`.

In the `List Get` section in Section A.3, we see that all results are significantly different with regards to the package energy.

6.3.2 List Removal

In this section we look at the efficiency of removing integers from different list types. The benchmarks are created using the same list types as in Section 6.3.1. In Listing 22 we see how the removal benchmark looks for `ImmutableList`, and the other list types' benchmarks are identical except for the type of list and their method for removing an element. The full benchmarks can be found on our git repository [103].

```
1  ...
2      [Benchmark("ListRemoval", "Tests removal from a
   ↪  ImmutableList")]
3      public static int ImmutableListRemoval() {
4          int result = 0;
5          ImmutableList<int> target = ImmutableList<int>.Empty;
6
```

```
7           for (int i = 0; i < LoopIterations; i++) {
8               for (int j = 0; j < Data.Count; j++) {
9                   target = target.Add(j);
10              }
11
12              for (int index = (Data.Count - 1) / 2; index >= 0;
                ↪  index--) {
13                  target = target.RemoveAt(index);
14              }
15
16              result += target.Count;
17          }
18
19          return result;
20      }
21  ...
```

Listing 22: The `ImmutableListRemoval` method which tests removal on an immutable list in the `ImmutableListBenchmarks` class.

In Listing 22 we show the `ImmutableListRemoval` benchmark created for testing removal of integers from an immutable list. We use two `for` loops inside the main `for` loop. In the first inner `for` loop we add 1000 integers to the immutable list called `target`. In the second inner loop we remove the first 500 integers in reverse order from the immutable list `target`. Because this list is immutable, we create a new immutable list each time we remove an integer. For non-immutable types, we do not create a new list each time.

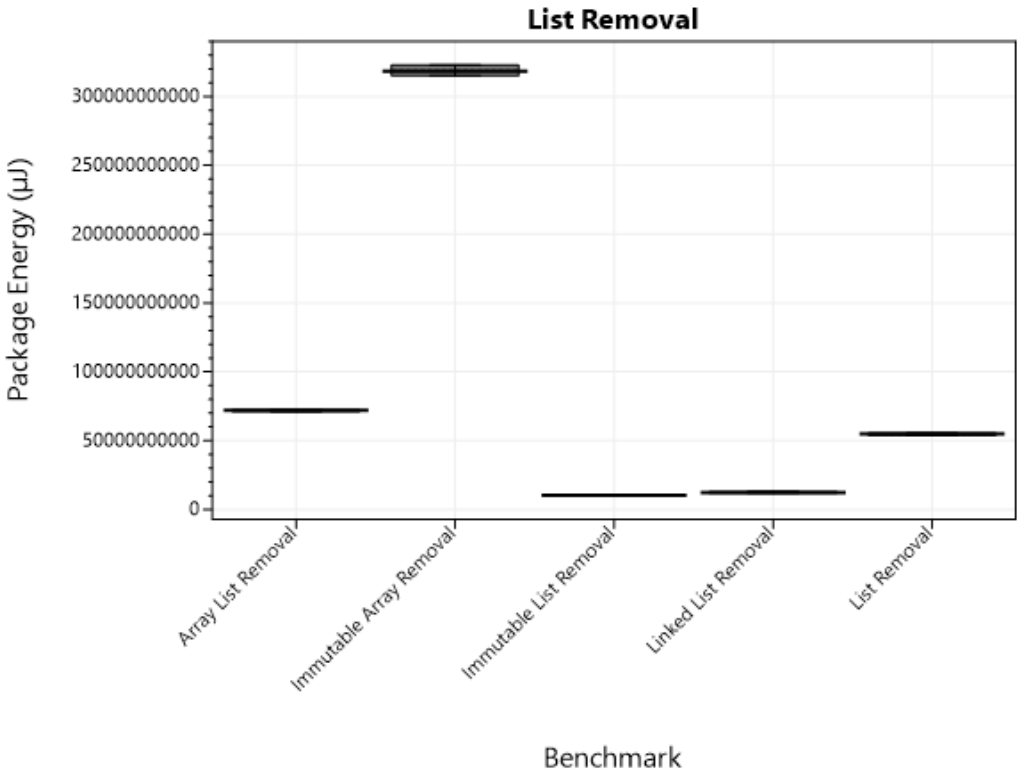

Figure 6.8: Boxplot showing how efficient different List Removal types are with regards to Package Energy.

In Figure 6.8 we see the package energy consumption for our benchmarks in the list removal group. The y-axis start from zero and spans around 350,000,000,000 µJ. We see that `Immutable List Removal` and `Linked List Removal` are the least package energy consuming benchmarks, we find this interesting as `Immutable List Removal` creates a new list and copies the old one over for every removal, which we assume would not be very energy efficient.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Array List Removal	4.117.686,666667	71.745.456.250,000	2.303.393.055,556
Immutable Array Removal	22.181.604,479167	318.774.679.166,667	12.531.503.125,000
Immutable List Removal	728.103,041858	10.353.928.913,417	464.979.773,509
Linked List Removal	852.388,915071	12.352.086.630,544	513.789.201,109
List Removal	3.400.574,174107	54.740.547.991,071	1.893.229.464,286

Table 6.7: Table showing the elapsed time and energy measurement for each List Removal.

In Table 6.7 we see the numeric results for time elapsed, package energy

consumption and DRAM energy consumption for the list removal benchmarks. We see that `Immutable List Removal` is both the fastest and the one consuming least energy of the benchmarks, with `Linked List Removal` as second. `List Removal` is more than four times slower than `Immutable List Removal` and consumes more than five times more package energy.

In the `List Removal` section in Section A.3, we see the *p*-values for package energy. All of the benchmarks are significantly different from each other.

6.3.3 Table Get

In this section we look at the efficiency of different methods of accessing integers from different table types. The table types we look at are:

- `Dictionary`
- `Hashtable`
- `ImmutableDictionary`
- `ImmutableSortedDictionary`
- `ReadOnlyDictionary`
- `SortedDictionary`
- `SortedList`

In Listing 23 we see an example of how the benchmarks are built. The differences between them are the type of table used and the method used for getting an element.

```
1  [Benchmark("TableGet", "Tests getting values sequentially from a
   ↪  Hashtable")]
2  public static int HashtableGet() {
3      int sum = 0;
4      for (int i = 0; i < LoopIterations; i++) {
5          for (int j = 0; j < Data.Count; j++) {
6              sum +=
                ↪  (int)Data[CollectionsHelpers.SequentialIndices[j]];
7          }
8      }
9
10     return sum;
11 }
```

Listing 23: The `HashtableGet` method which tests accessing items in a `Hashtable` in the `HashtableBenchmarks` class.

In Listing 23 we see the `HashtableGet` benchmark, which tests accessing items in the `Hashtable` called `Data`. Other benchmarks in this category use the same template, except `Data` is another table type.

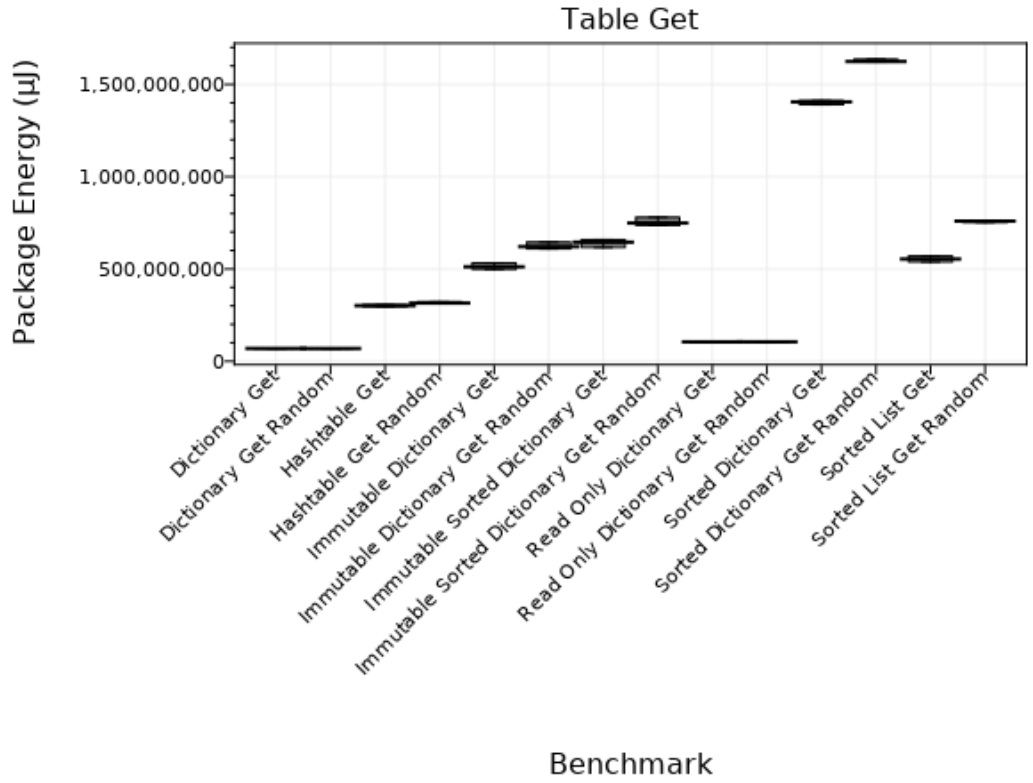

Figure 6.9: Boxplot showing how efficient different Table Get types are with regards to Package Energy.

In Figure 6.9 we see the package energy consumption for our benchmarks in the table get group. The y-axis start from zero and spans around 1.700.000.000 µJ. In Figure 6.9 we see that `Hashtable Get` consumes more than 4 times as much energy as `Dictionary Get`.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Concurrent Dictionary Get	1.072.901,458333	12.962.393.055,556	596.977.430,556
Concurrent Dictionary Get Random	1.072.031,818182	12.994.715.909,091	596.755.681,818
Dictionary Get	4.843,913710	69.332.938,639	2.695.199,924
Dictionary Get Random	4.844,832628	69.141.790,093	2.695.448,134
Hashtable Get	18.620,078823	300.895.396,205	10.472.509,766
Hashtable Get Random	19.502,656250	315.457.230,319	10.948.655,349
Immutable Dictionary Get	37.928,138428	512.774.717,204	21.094.523,112
Immutable Dictionary Get Random	50.475,181588	622.144.283,150	28.075.000,000
Immutable Sorted Dictionary Get	47.234,032411	645.217.337,943	26.275.274,493
Immutable Sorted Dictionary Get Random	60.901,821181	752.983.029,514	33.865.416,667
Read Only Dictionary Get	6.952,353516	104.625.971,680	3.865.769,531
Read Only Dictionary Get Random	6.937,612305	104.983.878,581	3.857.071,940
Sorted Dictionary Get	99.192,005208	1.403.638.151,042	55.151.388,889
Sorted Dictionary Get Random	114.615,112847	1.626.500.651,042	63.774.175,347
Sorted List Get	44.417,494081	552.496.978,575	24.705.158,026
Sorted List Get Random	63.285,692274	757.550.585,937	35.190.516,493

Table 6.8: Table showing the elapsed time and energy measurement for each Table Get.

In Table 6.8 we see the numeric results. Here the benchmarks for `Concurrent Dictionary` are also included, but were removed from Figure 6.9 as it made the graph unreadable. We can see in the results, that `Read only Dictionary Get` uses about 50% more energy than the `Dictionary Get`, which is surprising as we would expect the same performance from the two when accessing data.

In the `Table Get` section in Section A.3, we see that all but four results are significantly different with regards to the Package Energy. The four that are not significant are `Concurrent Dictionary Get` and `Concurrent Dictionary Get Random`, `Dictionary Get` and `Dictionary Get Random`, `Read Only Dictionary Get` and `Read Only Dictionary Get Random`, and `Immutable Sorted Dictionary Get Random` and `Sorted List Get Random`.

6.3.4 Findings

The results presented in this section can help a developer determine what constructs to use in different situations. We find that when having to use some type of list, and knowing that you will access elements from this list often, it is best to use an `array` if it fulfills the developers' need.

Next, we have the removal of elements from a list, in this case, we have observed that if energy consumption is the only thing being considered `ImmutableList` and `LinkedList` are the best candidates. Of course, there might be cases that while these are better for energy a developer will need to make compromises in which list is utilized.

In the last part, we saw how different types of tables performed when it came to getting an element. The table that showed the best performance was `Dictionary`, thereby the results suggest that if `Dictionary` fulfills a developer's needs it is the preferable type of table. Alternatively, `Hashtable` is the second-best option.

To summarize, the interesting findings in Collection are:

- `Immutable List get` consumes over 40 times more energy than the other benchmarks in the `List Get` group, except for `LinkedList`.
- `Linked List Removal` and `Immutable List Removal` consumes one third the energy of other list removal benchmarks.
- `Hashtable Get` and `Hashtable Get Random` consumes more than three times as much energy as `Dictionary Get` and `Dictionary Get Random`.
- `Read Only Dictionary Get` consumes more energy than `Dictionary Get`.

6.4 String Concatenation

In this section, we look at the efficiency of nine different approaches for concatenating strings:

- `Concat` (Using string.Concat)
- `Format` (Using string.Format)
- `Interpolation` (Using string interpolation $)
- `Interpolation Const` (Using string interpolation where strings are const)
- `Join` (Using string.Join)
- `Plus Comp Sign` (Using += multiple times)
- `Plus Comp Sign Single` (Using += and + on a single line)
- `Plus Sign` (Using +)
- `String Builder` (Using `StringBuilder`)

Each of these methods of concatenating strings achieve the same outcome using different language features. We start by looking at the benchmark `Interpolation`.

```
1  [Benchmark("StringConcat", "Tests string interpolation")]
2  public static string Interpolation() {
3      string str = "";
4      string iStr = "I ";
5      string am = "am ";
6      string a = "a ";
7      string stringStr = "string ";
8      string with = "with ";
9      string integer = "integer ";
10     int myInt = 42;
11     for (int i = 0; i < LoopIterations; i++) {
12         str = "";
13         str = $"{iStr}{am}{a}{stringStr}{with}{integer}{myInt}";
14     }
15
16     return str;
17 }
```

Listing 24: The `Interpolation` method which tests string interpolation in the `StringBenchmarks` class.

In Listing 24 we see the benchmark for testing string interpolation, and all other string concatenation benchmarks follow this form, by concatenating the strings on lines 3-9 and the integer on line 10.

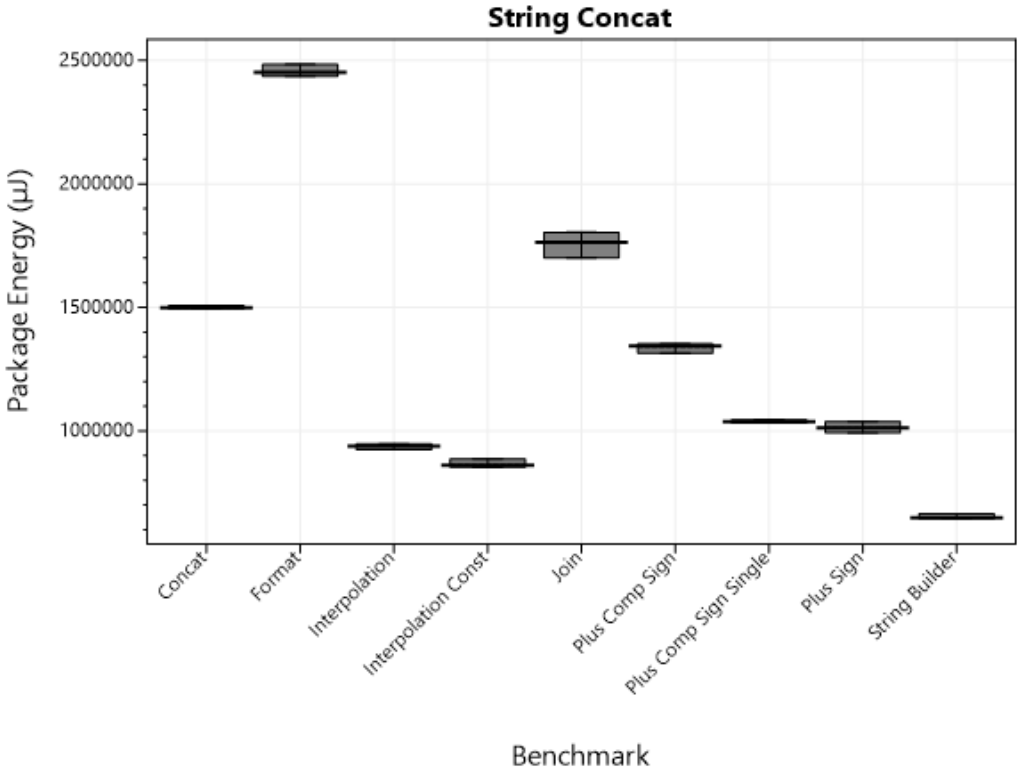

Figure 6.10: Boxplot showing how efficient different String Concatenation approaches are with regards to Package Energy.

In Figure 6.10 we see that the Package Energy consumption of the different string concatenation approaches varies widely. Note that the y-axis does not start from zero and spans 2.000.000 µJ. These results are surprising, as we expected that some of these approaches use the same or similar implementation under the hood, and apply the same optimizations. We expected that these approaches used similar implementations because they are similar and we expected that they can be used interchangeably without much effort.

The least energy efficient approach is using the `string.Format` approach, which uses approximately 2,5 times more energy compared to string interpolation or simply using + to concatenate the strings.

The most energy efficient approach is using a `StringBuilder`, as seen in Table 6.9 with an average consumption of 649.596,221 μJ.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Concat	105,095622	1.498.695,967	60.271,962
Format	168,237707	2.452.010,854	94.942,898
Interpolation	61,647413	937.183,656	34.811,974
Interpolation Const	59,775534	862.549,460	33.796,740
Join	117,485496	1.759.847,856	66.768,384
Plus Comp Sign	89,079091	1.339.160,840	51.656,262
Plus Comp Sign Single	72,462775	1.038.106,410	41.498,099
Plus Sign	69,501949	1.012.054,016	39.811,592
String Builder	42,375883	649.596,221	24.073,461

Table 6.9: Table showing the elapsed time and energy measurement for each String Concat.

In Table 6.9 we see the specific numeric results, and we can also extrapolate that the same pattern applies to the time taken by the benchmarks, as well as the DRAM energy consumption.

In the `String Concatenation` section in Section A.3 we have the table that contains the p-values for the benchmarks, from this we see that the difference in package energy consumption is significant for all of the results, which implies that the implementation of every approach is different.

6.4.1 Findings

These findings can then help determine what construct should be utilized, as most of these approaches can be simply swapped for another one. For instance, using `string.Format` can be easily replaced by string interpolation, which is way more efficient, both in terms of energy usage and execution time. However if energy is the only concern the results show that the optimal choice is `StringBuilder`. To summarize, the interesting findings are as follows:

- Approaches of concatenating strings vary in efficiency and speed for approaches that look similar, like `string.Format` and string interpolation,
- `StringBuilder` is significantly more efficient and faster than all other approaches.

6.5 Invocation

In this section, we look at the efficiency of different invocation methods. The section is split into two subgroups, one subgroup for `Local Function`, and the other subgroup for `Reflection`. The two subgroups are taken from the four subgroups in invocation, which can be seen in the `Invocation` section in Section A.3. The reason these two groups are the only ones selected for this section, is that we find they have interesting results.

Local Function

In this subsection we look at the efficiency of using local functions. We do this by creating four benchmarks. Two are static benchmarks where the local function is static and two non-static benchmarks. In the `Local Function` and `Local Static Function` we calculate and returns a value. For `Local Function Invocation` and `Local Static Function Invocation` another method is called which returns a value which we then return from our local function.

```
1  ...
2      private static readonly InvocationHelper InstanceObject =
       ↪ new();
3
4      [Benchmark("InvocationLocalFunction", "Tests invocation
   ↪   using an local function")]
5      public static int LocalFunction() {
6          int Calc() {
7              InvocationHelper.StaticField++;
8              return InvocationHelper.StaticField + 2;
9          }
10
11         int result = 0;
12
13         for (int i = 0; i < LoopIterations; i++) {
14             result += Calc() + i;
15         }
16
17         return result;
```

```
18      }
19  ...
```

Listing 25: The `LocalFunction` function tests invocation using a local function in the `LocalFunctionBenchmarks` class.

In Listing 25 we show the `LocalFunction` benchmark created for testing invocation. We use a `for` loop to iterate over a call to local function `Calc` and the initializer `i` which are added to the variable `result` for `LoopIterations` times. The `Calc` function is a local function within the benchmark, which increments a static field and then returns the int value of the static field plus 2. The other benchmarks in the `Local Function` group use the same general structure, except the local function, `Calc`, is different. In the `LocalFunctionInvocation` benchmarks invokes a method outside the local function, where the `LocalFunction` benchmark does the calculation in the local function itself. In the `static` benchmarks the local function is static.

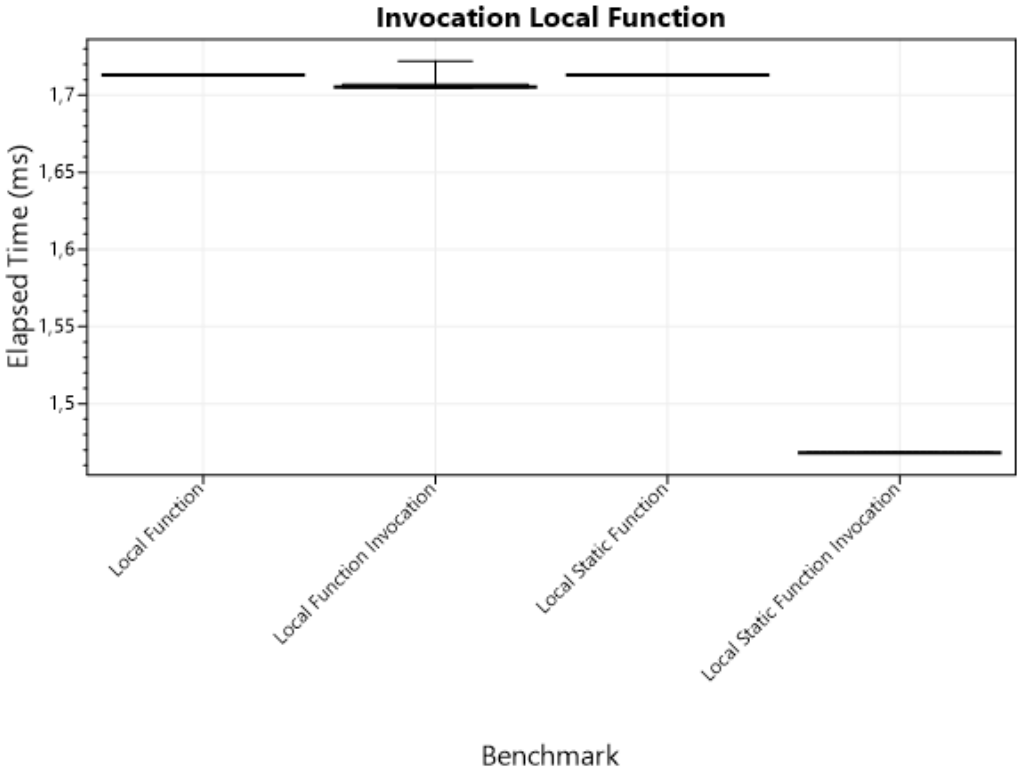

Figure 6.11: Boxplot showing how efficient different local function benchmarks are with regards to Elapsed Time.

In Figure 6.11 we can see the difference in time elapsed between the four `Local Function` benchmarks. The y-axis does not start from zero and spans approximately 0,3 ms. We can see that the `Local Static Function Invocation` has the lowest time elapsed, while `Local Function Invocation`, `Local Function` and `Local Static Function` are close together.

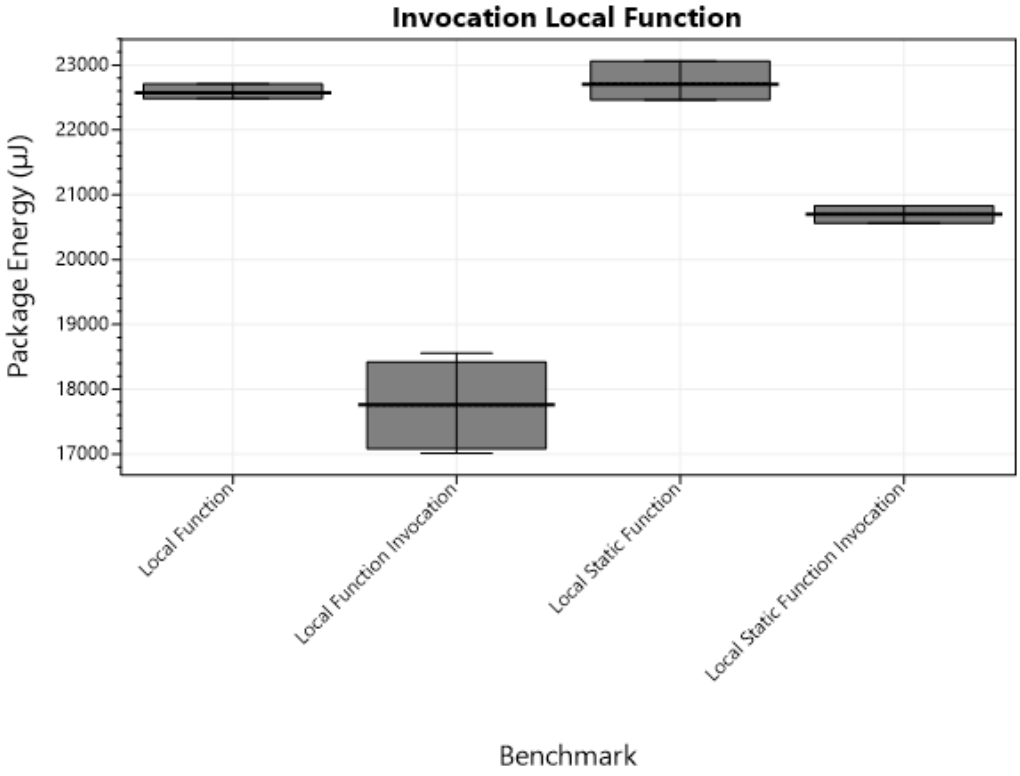

Figure 6.12: Boxplot showing how efficient different local function benchmarks are with regards to Package Energy.

In Figure 6.12 we can see the difference in package energy consumption between the four `Local Function` benchmarks. The y-axis does not start from zero and spans approximately 5,200 µJ. We can see that the `Local Function Invocation` benchmark consumes the least amount of energy, consuming less than `Local Function` and `Local Static Function Invocation`, which is interesting as `Local Function Invocation` has a higher elapsed time than `Local Static Function Invocation`. Furthermore, we saw that `Local Function Invocation` had almost as high time elapsed as `Local Function` and `Local Static Function`, but consumes about 20% less energy. We find this result interesting, as in most other results we have seen in our experiments, the elapsed time and energy consumption have a correlation, according to what is seen on graphs and measurement tables.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Local Function	1,713111	22.572,590	952,421
Local Function Invocation	1,705528	17.748,136	948,553
Local Static Function	1,713017	22.726,087	952,533
Local Static Function Invocation	1,468121	20.700,552	816,623

Table 6.10: Table showing the elapsed time and energy measurement for each Invocation Local Function.

In Table 6.10 we see the average results for time, package energy consumption and DRAM energy consumption for all benchmarks in the `Invocation Local Function` group. We can see again that `Local Static Function Invocation` is the fastest, while `Local Function Invocation` consumes the least Package energy. `Local Static Function Invocation` consumes the least DRAM energy while consuming the second least package energy, however it still consumes more combined energy than `Local Function Invocation`. `Local Function` and `Local static Function` almost have the same time elapsed, package energy and DRAM energy consumption.

In `local function` section of `Invocation` in Section A.3 in the table of elapsed time p-values for the benchmarks, we see that difference in time elapsed is significant for all of the results, except between `Local Function` and `Local Static Function`. We can see that the difference in time between `Local Function Invocation` and the others is significant.

In the package energy p-value table in the `local function` section of `Invocation` in Section A.3, we see the same as in the elapsed time p-value table, that the difference in package energy consumption is significant for all of the results, except between `Local Function` and `Local Static Function`. The difference in package energy consumption between `Local Function Invocation` and the others are significant, and the interesting results are analysed to find why the time and energy consumption does not correlate for `Local Function Invocation`.

Reflection

In this section, we look at the efficiency of using reflection.

In this subgroup we have 6 benchmarks testing method invocation using reflection. Where `Reflection` and `Reflection Static` tests invoking an instance method and a static method respectively. The flags benchmarks uses a parameter called `BindingFlags` to inform the run-time what we are looking for and what we intend to do with it. Examples of these are `Static`,

which means that static members are included in the reflection search. Another example is `InvokeMethod`, which specifies that the method for our reflection is supposed to be invoked, it can not be a constructor or an initializer [109]. Lastly we have delegate where we use a trick to help the compiler do more work before runtime.

```
...
    private static readonly MethodInfo MethodReflectionDelegate
    ↪ =
        typeof(InvocationHelper).
        ↪ GetMethod(nameof(InvocationHelper.Calculate),
            BindingFlags.Public | BindingFlags.Instance |
            ↪ BindingFlags.InvokeMethod);
...
    private static readonly Func<InvocationHelper, int>
    ↪ ReflectionDelegateInt =
        (Func<InvocationHelper,
        ↪ int>)Delegate.CreateDelegate(typeof(Func<InvocationHelper,
        ↪ int>),
            MethodReflectionDelegate);
...
    [Benchmark("InvocationReflection", "Tests invocation using a
↪ reflection on an instance method using delegate")]
    public static int ReflectionDelegate() {
        int result = 0;

        for (int i = 0; i < LoopIterations; i++) {
            result += ReflectionDelegateInt(InstanceObject) + i;
        }

        return result;
    }
...
```

Listing 26: The `ReflectionDelegate` function which tests invocation using a reflection on an instance method using delegate in the `ReflectionBenchmarks` class.

In Listing 26 we show the `ReflectionDelegate` benchmark created for testing invocation. We use a `for` loop to iterate over a reflection on an instance method `Calculate` using a delegate, which is done for `LoopIterations` times. In the `for` loop we add the result from `Calculate` and the initializer `i` to a variable called `result`. `Calculate` is a method in the `InvocationHelper` class. `Calculate` increments a field and then returns the field plus 2.

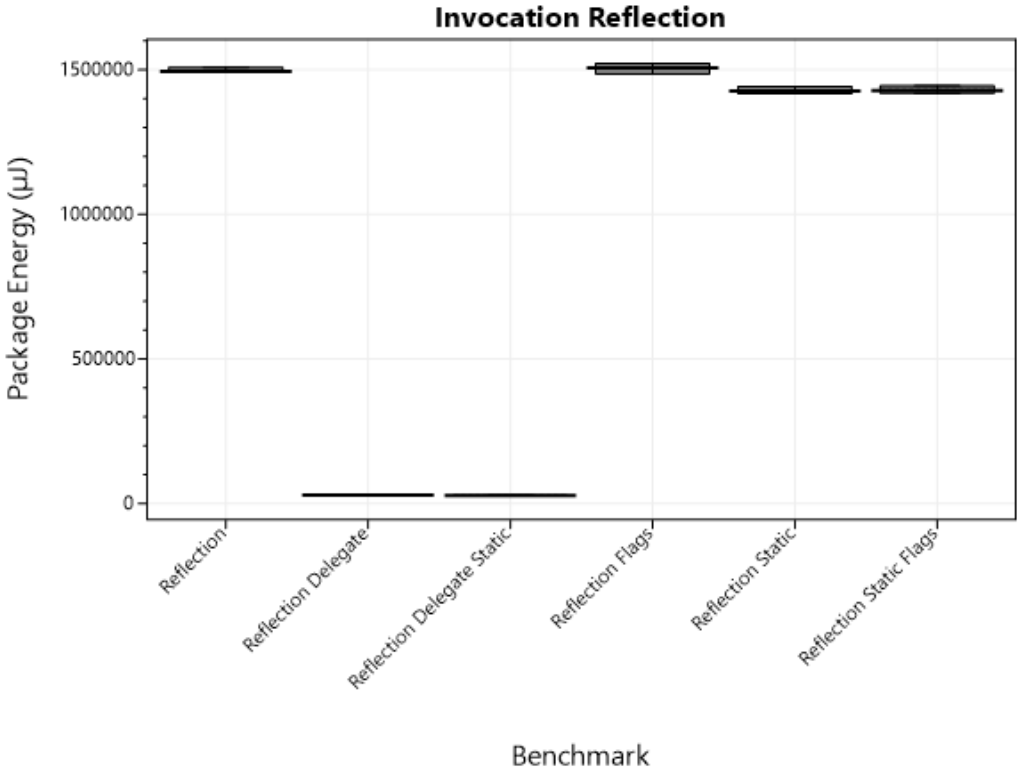

Figure 6.13: Boxplot showing how efficient different reflection benchmarks are with regards to Package Energy.

In Figure 6.13 we can see the difference in package energy consumption between the `Reflection` benchmarks. The y-axis start from zero and spans around 1.500.000 µJ. We can see that the `Reflection Delegate` and `Reflection Delegate Static` benchmarks consume considerably less energy than the other four benchmarks.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Reflection	97,219471	1.497.731,018	54.298,528
Reflection Delegate	2,202594	29.323,287	1.224,904
Reflection Delegate Static	1,958067	27.730,138	1.087,511
Reflection Flags	97,242559	1.506.979,540	54.267,544
Reflection Static	92,630929	1.429.110,845	51.727,083
Reflection Static Flags	92,815287	1.431.378,343	51.815,033

Table 6.11: Table showing the elapsed time and energy measurement for each Invocation Reflection.

In Table 6.11 we see the average results for time, package energy consumption and DRAM energy consumption for all the benchmarks in the `Reflection` group. We see that `Reflection Delegate` and `Reflection Delegate Static` consume around 2% of the package energy the other benchmarks consume. In addition we can see that `Reflection Delegate` and `Reflection Delegate Static` are more than 40 times faster than the other benchmarks. We look further into this difference in our analysis in our analysis.

In the `reflection` section in Section A.3, we see if our results in package energy between the benchmarks are significant. We can see that the results are significant except between `Reflection` and `Reflection flags`, and between `Reflection Static` and `Reflection Static Flags`.

6.5.1 Findings

In this section we have looked at two subgroups, namely `Local Functions` and `Reflection`.

As a developer when using local functions, we see that there is a split between which approach should be used depending on whether elapsed time or energy consumption is ranked as most important. If time is the primary concern, then `Local Static Function Invocation` seems to be the best choice, however if the developer is willing to compromise on time to improve energy consumption they can make us of `Local Function Invocation` as it proves best in energy consumption and second in elapsed time.

When it comes to `Reflection` the result show that using `Delegates` is the best option, both in regards to time and energy consumption. As such it is preferable to use `Delegates` when possible, and if the `Delegate` can be `static`, there can be even further improvements.

To summarize, the interesting findings in Invocation are:

- Time and Package Energy consumption do not seem to correlate for `Local Function Invocation`, this is interesting to us, as it is contrary to most of our results.
- Delegate using reflection is faster and consumes less energy than just using reflection, using reflection with flags or using static reflection.

6.6 Objects

In this section, we look at the efficiency of different object invocation methods. C# offers four ways of expressing objects in which are:

- `class`
  Reference type
- `struct`
  Value type
- `record`
  Immutable reference type
- `record struct`
  Immutable value type

We look at the invocation of static and instance methods for each of these four types of objects, the benchmarks follow the structure seen in Listing 27, substituting the way of expressing an object between the benchmarks.

```
...
    [Benchmark("ObjectInvocation", "Tests invocation of a method
↪   on a class")]
    public static int ClassMethod() {
        int result = 0;
        ClassHelper classObject = new ClassHelper();

        for (int i = 0; i < LoopIterations; i++) {
            result += classObject.Calculate() + i;
        }

        return result;
    }
...
```

Listing 27: The `ClassMethod` method which tests object invocation using a method in the `LocalFunctionBenchmarks` class.

In Listing 27 we see the `ClassMethod` benchmark created for testing object invocation. We use a `for` loop to iterate over a method `Calculate` and the initializer `i` which are added to the variable `result` for `LoopIterations` times. The `Calculate` method is in the `ClassHelper` class, it increments a variable `field` by one, then returns `field` plus two. The `Class Method Static` benchmark has the same structure, but has a static method instead of the `Calculate` method. The full benchmarks can be seen in our git repository [103].

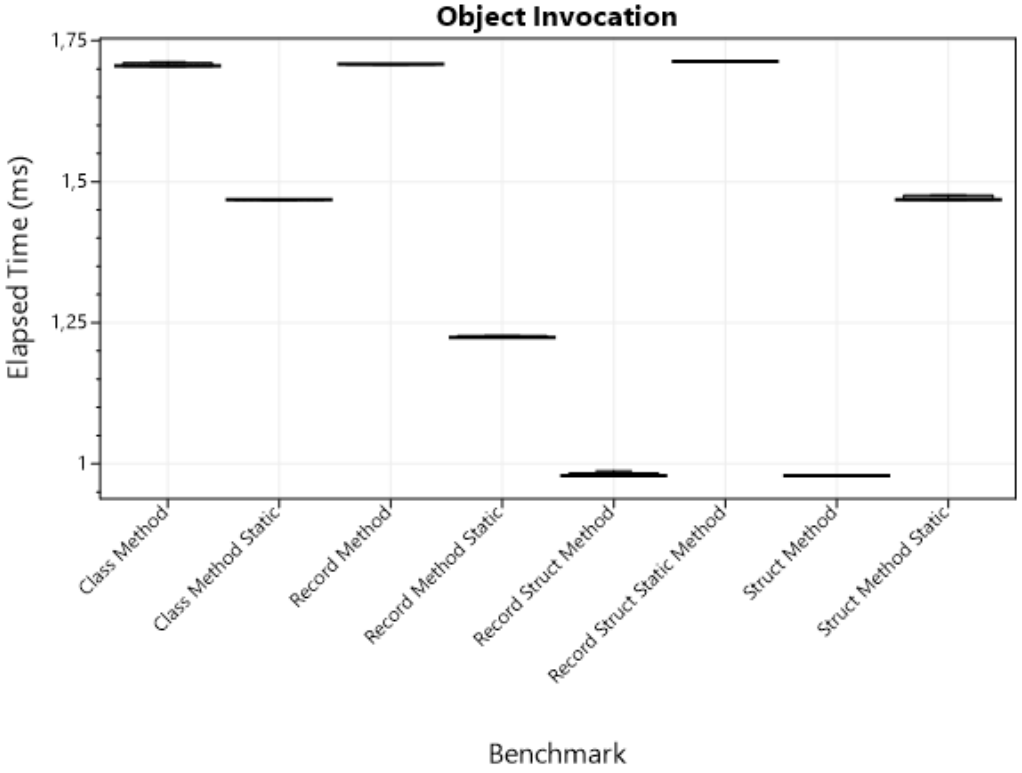

Figure 6.14: Boxplot showing how efficient different Object Invocation types are with regards to Elapsed Time.

In Figure 6.14 we see the elapsed time, for the benchmarks in our object invocation grouping. The y-axis does not start from zero, and only spans around 0,8 ms. We see that `Class Method` has a higher elapsed time than `Class Method Static`.

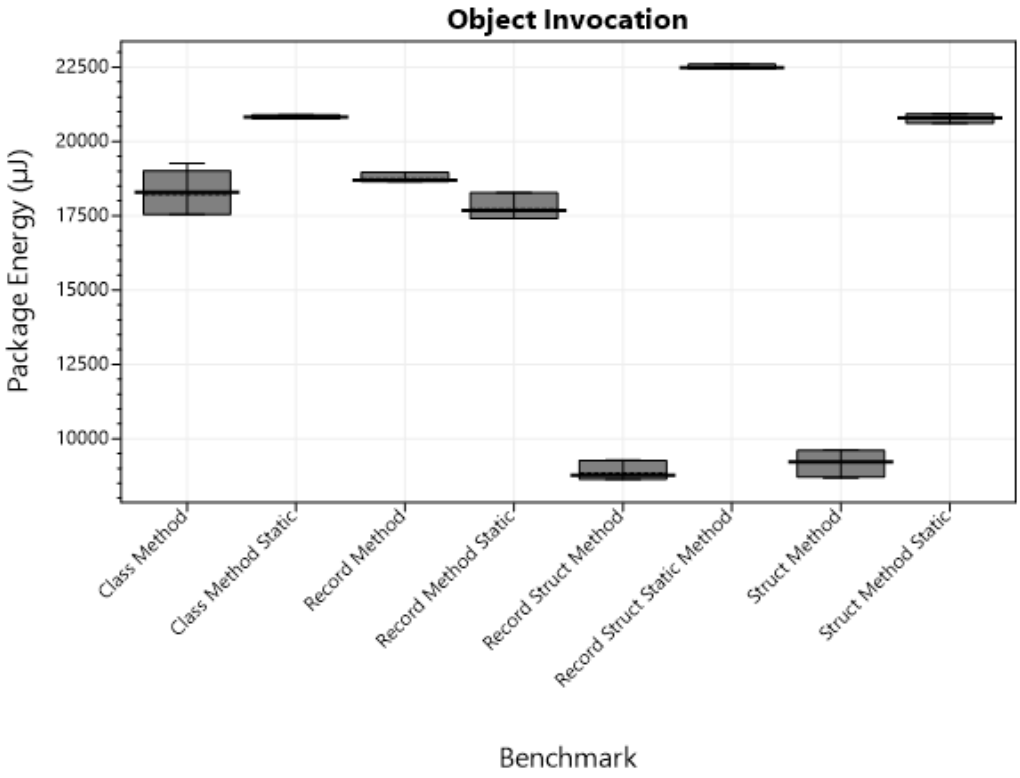

Figure 6.15: Boxplot showing how efficient different Object Invocation types are with regards to Package Energy.

In Figure 6.15 we see the package energy consumption for the benchmarks. The y-axis does not start from zero, and only spans around 13,000 µJ. We see that `Class Method` consumes less energy than the static variant. This is interesting as, most of our results show that time and energy consumption correlate.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Class Method	1,705729	18.222,192	948,875
Class Method Static	1,468121	20.817,694	816,381
Record Method	1,708151	18.740,392	950,319
Record Method Static	1,223811	17.721,990	681,006
Record Struct Field Method	1,713339	22.498,618	952,704
Record Struct Method	0,979123	8.842,624	544,515
Struct Method	0,978953	9.201,011	544,441
Struct Method Static	1,468980	20.776,040	818,018

Table 6.12: Table showing the elapsed time and energy measurement for each Object Invocation.

In Table 6.12 we see the numeric results for time elapsed, package energy consumption and DRAM energy consumption for the benchmarks. In the table we see again that `Class Method` is slower than `Class Method Static`, while consuming less package energy. `Class Method` does consume more DRAM energy than `Class Method Static`, but overall `Class Method` consumes the least energy of the two.

In the `invocation` section of the `Objects` section in Section A.3, we can see the p-values for elapsed time between the benchmarks. We can see that there is a significant difference between `Class Method` and `Class Method Static`.

In the `invocation` section of the `Objects` section in Section A.3, we can see the p-values for package energy consumption between the benchmarks. We can see that there is a significant difference between `Class Method` and `Class Method Static`. We can conclude that `Class Method` and `Class Method Static` have different elapsed time with `Class Method` having the highest, while they also have different package energy consumption where `Class Method Static` has the highest consumption.

6.6.1 Findings

While this section shows us the performance of invocation of an object and how the performance is affected by different representations, it does not show other operations like accessing a field from the object. This must be kept in mind when evaluating what representation to use. What the result does show is that for a developer whose primary concern is energy consumption they should use `structs`. One should also consider what is more important when using classes, time, or energy? As static class methods

decrease the time needed but cause an increase in energy consumption.

To summarize, the interesting finding in Object Invocation is:

- `Class Method` has a higher elapsed time than `Class Method Static`, while it consumes less energy.

6.7 Inheritance

In this section, we examine the efficiency of different types of inheritance.

We have created six benchmarks to test different types of inheritance and how they compare to one another. The benchmarks are built in such a manner that first a type of inheritance is used to create an object and then the object is invoked to update and return a value. In the `Abstract` benchmark our object has inherited from an abstract class. The `Inheritance` benchmark has our object created using regular inheritance. The benchmark for `Inheritance Virtual` is seen in Listing 28, this structure is also used for the other benchmarks. In this case, a class is used for our object has a virtual method. For the benchmark `Inheritance Virtual Override` we have inherited from the class from `Inheritance Virtual` and overridden the method for updating and returning a value. With the `Interface` benchmark we have inherited from an interface to create to class for our object. As the last benchmark, we have `No Inheritance`, where there has been no inheritance of any kind and instead the original class has been used to create our object.

```
...
    [Benchmark("Inheritance", "Tests getting and updating a
↪   value using a virtual method")]
    public static int InheritanceVirtual() {
        int result = 0;
        VirtualHelper helper = new VirtualHelper();
        for (int i = 0; i < LoopIterations; i++) {
            result += helper.UpdateAndGetValue();
        }

        return result;
    }
...
```

Listing 28: The `InheritanceVirtual` method which tests inheritance using virtual methods in the `VirtualHelper` class.

In listing Listing 28 we see the `InheritanceVirtual` benchmark created for testing the use of virtual methods when getting and updating a value. We use a `for` loop to iterate over a virtual method `UpdateAndGetValue` which increments a `field` variable and returns it.

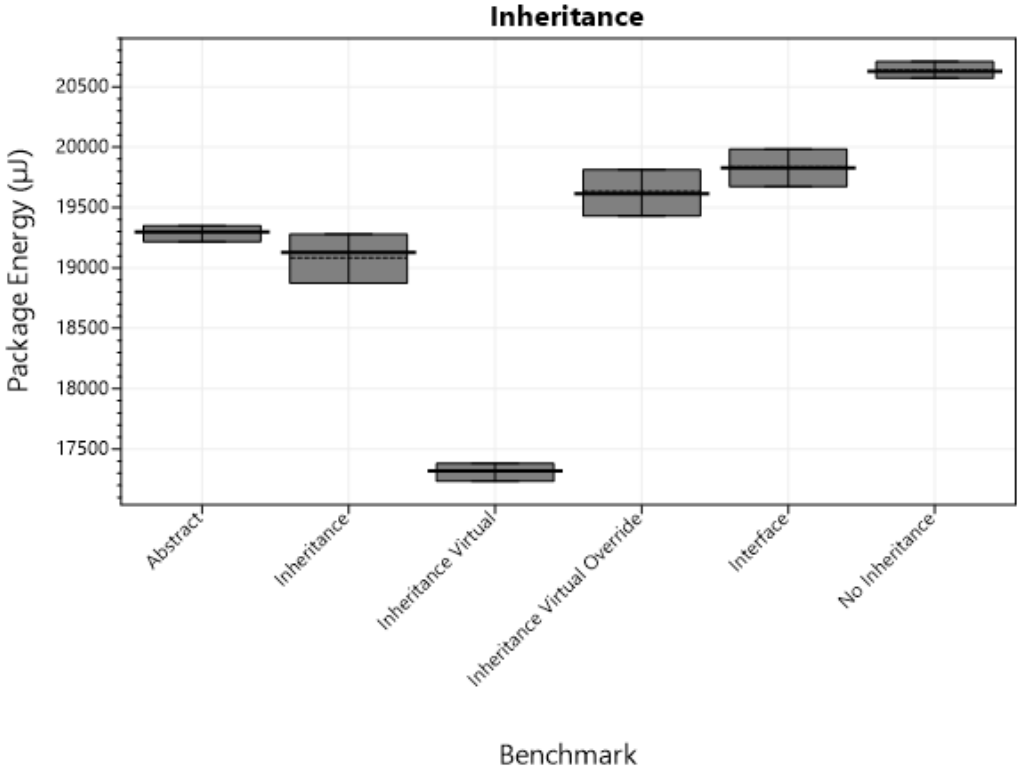

Figure 6.16: Boxplot showing how efficient different Inheritance types are with regards to Package Energy.

In Figure 6.16 we see the package energy consumption for our inheritance benchmarks. The y-axis does not start from zero and spans around 4.000 µJ. We see that `Inheritance Virtual` consumes the least package energy, while `No Inheritance` consumes the most of the benchmarks. We find this interesting as we assume virtual inheritance consumes the same amount of energy as inheritance, while no inheritance consumes the same or less energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Abstract	1,467985	19.300,014	816,340
Inheritance	1,468008	19.082,808	816,461
Inheritance Virtual	1,223562	17.313,036	680,684
Inheritance Virtual Override	1,468085	19.632,412	816,560
Interface	1,468055	19.834,493	816,239
No Inheritance	1,713479	20.634,361	952,703

Table 6.13: Table showing the elapsed time and energy measurement for each Inheritance.

In Table 6.13 we see the numeric results for time elapsed, package energy consumption and DRAM energy consumption. We see that `Inheritance Virtual` is the fastest benchmark and consumes the least package and DRAM energy. `No inheritance` is the slowest and consumes the most package and DRAM energy.

In the `Inheritance` section in Section A.3 we can see the p-values for package energy consumption between the benchmarks. We can see that all the results have a p-value lower than 0,05. This tells us that significant differences exist between all the benchmarks.

6.7.1 Findings

This section gives a good insight into the type of inheritance a developer should use when the situation allows for it. We see that the `Inheritance Virtual` benchmark has the best performance across both time and energy consumption, also implying that virtual inheritance is the best option for a developer regardless of whether they are more concerned with time or energy. Moving to the results of the other types of inheritance are relatively close to one another, with `No Inheritance` being the worst.

To summarize, the interesting findings in Inheritance are:

- `Inheritance virtual` is more efficient compared to other types of inheritance, and
- `No Inheritance` is the least efficient.

6.8 Exceptions

In this section we look at the efficiency of handling exceptions. This section is split into two parts, first looking at exception handling where exceptions occur, and afterwards where exceptions do not occur.

6.8.1 `Try catch` with exception

Here in the first group within `Exceptions` we look at five different ways that `Try catch` can be used for catching exceptions.

The first benchmark called `Try Catch All E`, uses a `try catch` which will catch all exceptions. We call the second benchmark `Try Catch EW Message`, in this benchmark an `ArgumentException` along with a error message is manually thrown. The third benchmark `Try Catch EWO Message` is identical to `Try Catch EW Message`, however no message accompanies the thrown exception. In Listing 29 we see how the benchmark called `Try Catch Specific E` functions. We add a benchmark called `Try Catch E With if`, which imitates the use of a `try catch` through the use of `if` and `else`.

These benchmarks allows us to determine how `try catch` performs as a control structure, both with and without specific exceptions thrown, as well as accompanying messages compared to one another, we also see how it compares to an equivalent structure using `if` and `else` instead of `try catch`. The full benchmarks can be found in our git repository [103].

```
1  [Benchmark("Exception",
2      "Tests try-catch exception thrown with specific catch
   ↪  statement, throws floor(LoopIterations / 2) exceptions")]
3  public static int TryCatchSpecificE() {
4      int a = 1;
5      int b = 2;
6      for (int i = 0; i < LoopIterations; i++) {
7          try {
8              a *= 2;
9              a += 2;
10             a %= 20;
11             a /= b;
12             b = 0;
13         }
14         catch (DivideByZeroException) {
15             a %= 10;
16             b = 2;
17         }
18     }
19     return a;
```

```
20 }
```

Listing 29: The `TryCatchSpecificE` method which tests catching the `DivideByZeroException` in the `ExceptionBenchmarks` class.

In Listing 29 we see a benchmark that tests a `try catch` statement, where an exception is thrown every second iteration, and the statement specifically catches `DivideByZeroException`.

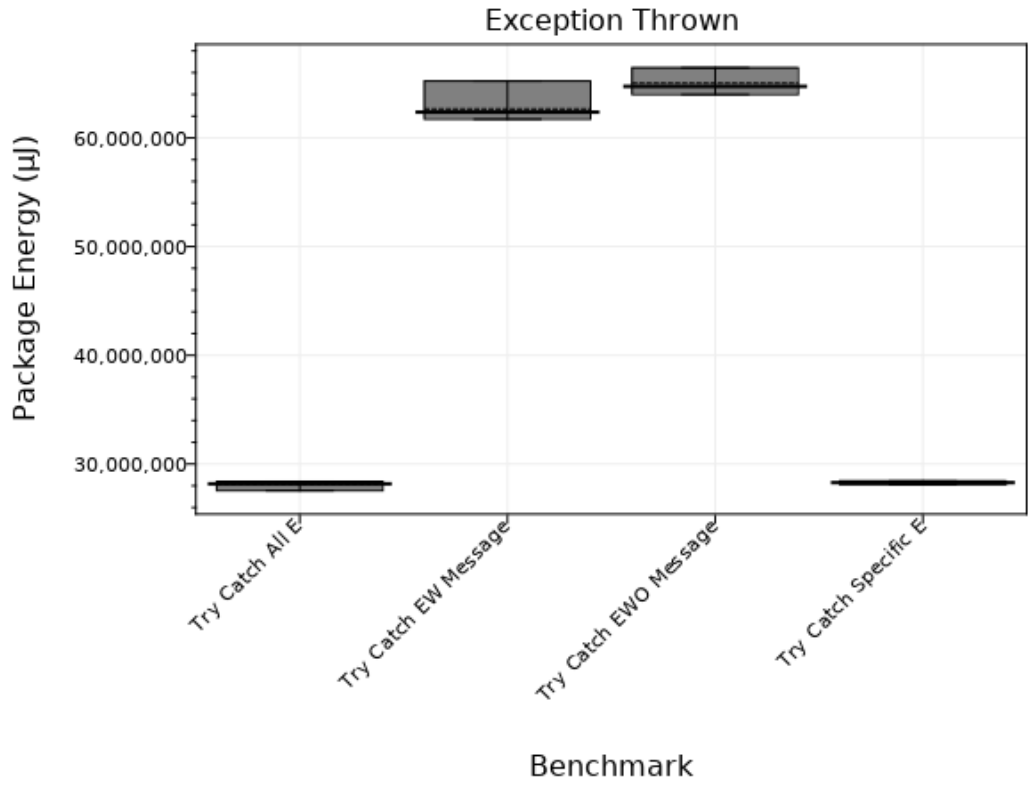

Figure 6.17: Boxplot showing how efficient different Exception Thrown types are with regards to Package Energy.

In Figure 6.17 we see the Package Energy consumption of, among others, the benchmark in Listing 29. We see in the figure that the y-axis does not start from zero and spans around 41.000.000 µJ. The `Try Catch EW Message` throws an `ArgumentException` with a string message, and the `EWO` version throws the same exception without a message.

It is surprising that throwing an `ArgumentException` uses about twice

as much energy as causing a `DivideByZeroException` as in Listing 29, even though the same amount of exceptions are thrown.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Try Catch All E	2.005,725403	28.131.047,363	1.118.590,698
Try Catch EW Message	4.347,888499	62.661.289,141	2.424.532,983
Try Catch EWO Message	4.484,282973	65.027.300,347	2.499.001,058
Try Catch Specific E	1.988,904690	28.280.468,750	1.109.098,985
Try Catch E With if	7,161153	64.449,111	3.982,336

Table 6.14: Table showing the elapsed time and energy measurement for each Exception Thrown benchmark.

In Table 6.14 we see the numeric results, with an extra benchmark called `Try Catch E With if`, which simply uses an `if` statement to check if a `DivideByZeroException` occurs. The `if` approach is about 280 times faster and 440 times more energy efficient with regards to Package Energy than using a `try catch` block when an exception occurs every second iteration.

In the `Exceptions` section in Section A.3, we see the p-values for the five benchmarks with regards to Package Energy. We see there is not a significant difference between catching a specific exception versus catching the general `Exception`.

6.8.2 `Try catch` without exception

Here in the second group within `Exceptions` we look at three different ways that `Try catch` behaves if no exceptions are actually thrown.

The first benchmark we call `Try Catch E With if`, which imitates the use of a `try catch` through the use of `if` and `else`. The second benchmarks uses a try `try catch`, where the `throw` statement does not throw anything, we call this `Try Catch No E`. Finally we have `Try Finally Equiv No E` which has the same code as `Try Catch No E`, except it is created without a `try catch`. It should be noted, that because an exception has to occur in `Try Catch E With if`, slightly more code is executed in this benchmark and they are not equivalent, however, it is included in this comparison as we find it surprising that it is almost as efficient as a `try catch` block where no exceptions are thrown and less code is executed.

Testing these benchmarks allows us to determine how `try catch` performs as a control structure compared to the equivalent use of `if` and no `try catch` like structure, in cases where there are no exceptions.

```
[Benchmark("Exception", "Tests try-catch no exception thrown")]
public static int TryCatchNoE() {
    int a = 1;
    int b = 2;
    for (int i = 0; i < LoopIterations; i++) {
        try {
            a *= 2;
            a += 2;
            a %= 20;
            a /= b;
        }
        catch (Exception) {
            a %= 10;
            throw;
        }
    }
    return a;
}
```

Listing 30: The `TryCatchNoE` method which tests a `try catch` block that does not throw an `exception` in the `ExceptionBenchmarks` class.

In Listing 30 we can see a benchmark that tests a `try catch` statement, where no exception is thrown.

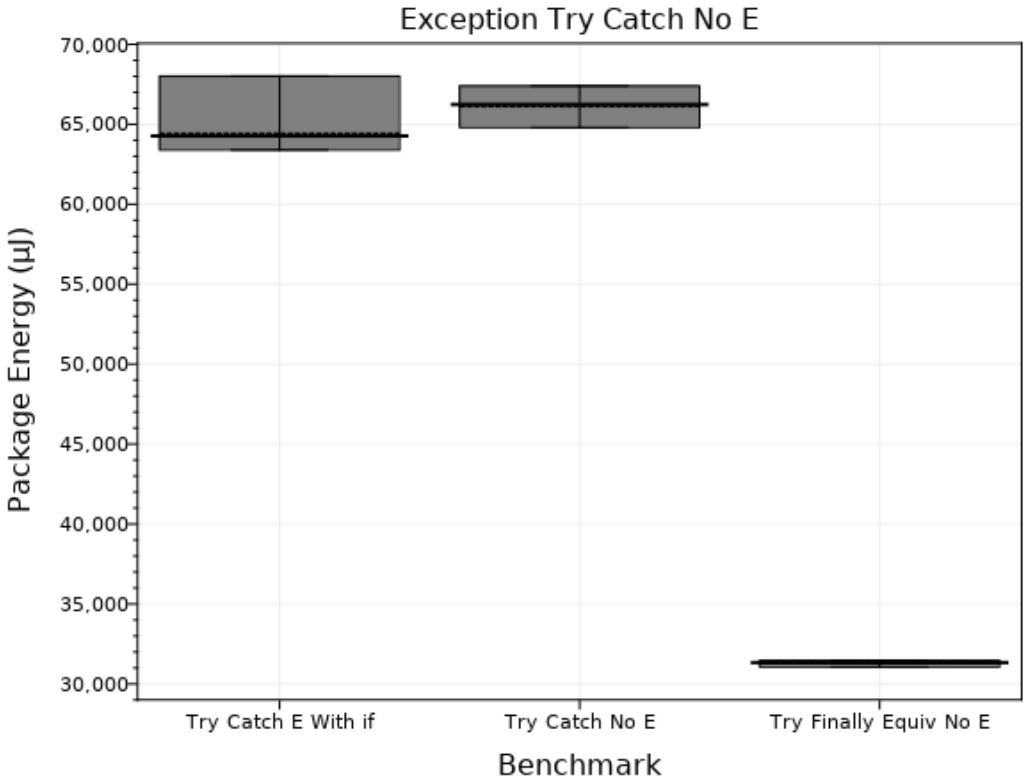

Figure 6.18: Boxplot showing how efficient different exception handling methods are with regards to Package Energy.

In Figure 6.18 we see the package energy consumption for three benchmarks. The y-axis does not start from zero and spans around 40.000 µJ.

We find it interesting that the package energy consumption increases with approx 110% when taking the code in `Try Finally Equiv No E` and putting it in a `try catch` block, as seen in Section A.3. This also goes against what [4] found in Java, namely that there was no extra overhead of wrapping code in a `try catch` block if no exceptions are thrown.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Try Catch E With if	7,161153	64.449,111	3.982,336
Try Finally Equiv No E	3,546218	31.294,703	1.970,243
Try Catch No E	7,305439	66.096,315	4.062,883

Table 6.15: Table showing the elapsed time and energy measurement for each Exception Try Catch No E.

In the `Exceptions` section in Section A.3, we see that the difference be-

tween all the results in this group are significant with regards to package energy consumption.

6.8.3 Findings

The finding presented in this section help a developer determine how to handle exceptions as well as showing how `try catch` can be used as a control structure.

In the first group we saw how `Try Catch Specific E` and `Try Catch All E` were best in regards to energy when it came to handling exceptions, so in a where a developer know exceptions can occur and have no need for messages to accompany them, these are preferable. However to note here is that if you can use `if` instead of `try catch` it is very much the preferred choice.

In the second group we have seen how if no exceptions are thrown is better to avoid the usage of `try catch`, showing that it is not a good choice for controlling the flow of a program, when it is not needed.

To summarize, the interesting findings are as follows:

- Using a `try catch` statement when `exceptions` are thrown 50% of the time, is 440 times slower than checking for the `exception` condition using an if statement,
- Throwing an `ArgumentException` uses about twice as much energy as causing a `DivideByZeroException`, and
- Wrapping code in a `try catch` statement increases package energy consumption with approximately 110% when no `exceptions` are thrown.

6.9 Summary

In this chapter, we have looked at a selection of results. We chose these results as they were interesting or unexpected. In short, the following results were interesting or unexpected:

- Unsigned integer datatypes are at least 1% more efficient than signed integer datatypes, except
- `ushort` is only 1% more efficient than `short`,

- `uint` is the most efficient datatype,
- `switch` is more efficient than `if` statements when applicable,
- Conditional operators are more efficient than `if` statements when looking at package energy,
- `if` consumes the least amount of energy while being the slowest of the benchmarks it is compared to,
- `Immutable List get` consumes over 40 times more energy than the other benchmarks in the `List Get` subgroup, except for `LinkedList`,
- `Linked List Removal` and `Immutable List Removal` consumes one third the energy of other list removal benchmarks,
- `Hashtable Get` and `Hashtable Get Random` consumes more than three times as much energy as `Dictionary Get` and `Dictionary Get Random`,
- `Read Only Dictionary Get` consumes more energy than `Dictionary Get`,
- Approaches of concatenating strings vary in efficiency and speed for approaches that look similar, like `string.Format` and string interpolation,
- `StringBuilder` is significantly more efficient and faster than all other approaches,
- Time and package energy consumption does not seem to correlate for `Local Function Invocation`, this is interesting to us, as it is contrary to most of our results,
- Delegate using reflection is faster and consumes less energy than just using reflection, using reflection with flags or using static reflection,
- `Class Method` has a higher elapsed time than `Class Method Static`, while it consumes less energy,
- `Inheritance virtual` is more efficient compared to other types of inheritance,
- no inheritance is the least efficient,

- Using a `try catch` statement when the `DivideByZeroException` is thrown 50% of the time, is 440 times slower than checking for the `exception` condition using an if statement,
- Throwing an `ArgumentException` uses about twice as much energy as causing a `DivideByZeroException`, and
- Wrapping code in a `try catch` statement increases package energy consumption with approximately 110% when no `exceptions` are thrown.

We analyse the results, attempting to find the cause of the benchmarks' behaviours in Chapter 7.

Chapter 7

Results Analysis

In this chapter, we analyse the results presented in Chapter 6. We use a four step analysis process created by us, with the goal of explaining the results of the different benchmarks in their respective groups. We divide the benchmark groups into sections in which we analyse the interesting and unexpected benchmark results.

7.1 Process

To analyse the results we go through the following four steps:

1. See if there is a difference between the benchmarks that explains the difference in performance, as sometimes it has been necessary to create slightly different benchmarks, to ensure that they are not optimized away by the compiler,
2. read the documentation for the language constructs we are looking at to see if there is an explanation for differing energy consumption,
3. look at the Intermediate Language (IL) code to see if there is a difference between the benchmarks,
4. look at the assembly code in debug mode to see if there is a difference between the benchmarks.

First, we compare the C# source code for the benchmarks to check for obvious differences.

Secondly, we look at the documentation that Microsoft has provided at [110], otherwise, we use the Common Language Infrastructure Standard [111].

In the third step, we use the IDE Rider [112] since it allows us to see the generated IL code by using the IL Viewer window [113]. It is also possible to use different IDEs such as Visual Studio which also has an extension to view IL code. Since we have used Rider as our IDE throughout the project we are familiar with it and have therefore chosen to use it to view the IL code.

In the fourth step, we use Visual Studio's [114] Disassembly window [115], which shows the assembly code generated by the Just-In-Time (JIT) compiler. We have chosen to use Visual Studio since Rider does not have this feature and we have experience with Visual Studio. This tool can only be used when address-level debugging is enabled which it is in Debug mode [115], this is why we do not look at the assembly code in release mode.

If these steps do not result in an explanation, we save further analysis for future work in Chapter 10.

The full analysis, of the different benchmark results, is not included in this chapter. The full analysis for the results can be found in Section A.4.

7.2 Primitive Integer

The first results we look at are the primitive integer results, specifically:

- Unsigned integer datatypes are at least 1% more efficient than signed integer datatypes, except
- `ushort` is only 1% more efficient than `short`, and
- `uint` is the most efficient datatype.

We start by analyzing why unsigned integer datatypes are more efficient than signed integer datatypes.

7.2.1 Unsigned vs Signed Integer Datatypes

To get an explanation for why unsigned integer datatypes are at least 1% more efficient than signed integer datatypes we look at the code for the benchmarks. The only difference we find in the benchmarks is that they use

different datatypes, this is not enough to explain why there is a difference, therefore we go to the next step, which is looking at documentation.

The documentation [116] does not give any explanation for why there is such a large difference in the results, therefore we go to the third step which is looking at the IL code.

When looking at the IL code, we find a larger difference than before. Here we find that the signed integer division and modulo uses the `div` and `rem` instructions, while the unsigned integer division uses the `div.un` and `rem.un` instructions.

When looking at [111, p.356-357, p.381-382] there is one interesting part for these instructions, namely that the instructions `div` and `rem` work for both integer and floating point datatypes, while `div.un` and `rem.un` only are present for unsigned integer datatypes.

This could be the reason why there is a difference between signed and unsigned datatypes. To make sure this is the case, we create the relevant benchmarks but without the division and modulo operations, these benchmarks give the same results with p-values above 0,05 meaning that this is the reason behind why there is a difference. The full analysis is seen in Section A.4.

7.2.2 Short Integer Datatypes

There is one case where an unsigned integer datatype is less than 1% more efficient than a signed integer datatype, the `ushort` datatype compared with the `short` datatype.

Here we go through the steps again, the code for the benchmarks only differ in the datatype, which is not enough to explain the difference. We look in [111, p.317], here we find why `short`, `ushort`, `byte` and `sbyte` are less efficient compared to the other integral datatypes, as it states "Loading from these locations onto the stack converts them to 4-byte ...", meaning these datatypes are converted to 4-byte (32-bit) integers, which means additional operations occur in the background. This also explains why there is a difference in how large the difference in performance is between the signed and unsigned variants of these datatypes, as they are turned into signed 4-byte integers in all cases, and therefore `div.un` and `rem.un` instructions may not be used.

To confirm that these instructions are not used, we look at the IL code, where we find that the `div` and `rem` instructions are used for all four of the datatypes (`short`, `ushort`, `byte` and `sbyte`). This explains why the differ-

ence between `short` and `ushort` is so small, however there is still quite a large difference between `byte` and `sbyte`, which is unexplained. According to [111, p.20] Table I.1, `byte` is a Common Language Specification (CLS) type, while `sbyte` is not, meaning `byte` has more requirements compared to `sbyte`.

We can not verify that this is the reason for the difference in performance, therefore we look at the assembly code for `sbyte` and `byte`. Here we find that the only difference between the two benchmarks is that `sbyte` uses the `movsx` instruction while `byte` uses the `movzx` instruction. The `movsx` instruction extends the given variable to a 32-bit integer with the same signed bit as the original variable (meaning if it is negative, the converted variable is also negative) while the `movzx` instruction extends the given variable to a 32-bit integer with zero-extension, meaning all the unused bits are 0's meaning it will always be a positive integer after converting. This does not give an explanation besides the `movsx` instruction being less efficient than the `movzx` instruction, therefore we leave further exploration of this result to future work in Chapter 10. The full analysis process is included in Section A.4.

7.2.3 Unsigned Integer

The result that `uint` is more efficient compared to all other integer datatypes is interesting, as the computer runs 64-bit, and we therefore expect `ulong` to be more efficient.

We look at the benchmarks, here we find no difference besides the datatype, which is not enough to explain the difference. There is no information in [116] that explains the results seen, therefore we go directly to the IL code.

The only difference in the IL code is that the instruction `conv.i8` occurs for all the `ulong` operations, this difference is expected, and not enough to explain the difference in result. Therefore, we look at the assembly code for these benchmarks.

When looking at the assembly code, it is clear to see why there is a difference in performance. The differences are as follows:

1. The addition and subtraction uses the `inc` and `dec` instructions for `uint`, while for `ulong`, 1 is added and subtracted,

2. The multiplication uses the `lea` instruction for `uint`, while using `imul` for `ulong`,

3. The division uses the `shr` instruction for `uint`, while using `xor` and `div` for `ulong`, and

4. The `movsxd` instruction is used for all the `ulong` operations.

These differences show that operations with `uint` are optimized by the compiler in a lot of cases. We therefore conclude that this is why `uint` is more efficient than other unsigned integer datatypes. For a look at all steps of the analysis process see Section A.4.

7.2.4 Summary

Over the course of this section, we find that the reason for unsigned integer datatypes being 1% or more efficient compared to signed integer datatypes is because unsigned division and modulo are more efficient compared to the signed variant.

The exception to this is the `short` and `ushort` integer datatype, where the unsigned division and modulo instructions are not used because they are converted to a signed 32-bit integer during the operations. However, the `byte` and `sbyte` integer datatypes do the same, despite still having a larger difference between them. It was not possible to find conclusive evidence to why `byte` is more efficient compared to `sbyte`, however `byte` is a CLS type while `sbyte` is not, so `byte` has more requirements compared to `sbyte` which could be the reason.

Lastly, the `uint` datatype is the most efficient because the compiler has several ways to optimize operations when using a `uint` compared to `ulong` and `unint`.

7.3 Selection

Next we look at the selection results, as these also had some interesting findings, which are:

- `switch` is more efficient than `if` statements when applicable,
- conditional operators are slightly more efficient than `if` statements when looking at package energy, and
- `if` consumes the least amount of energy while being the slowest of the benchmarks compared to the `if else` and `if else if` benchmarks.

We start by analyzing why `switch` statements are more efficient than `if` statements when applicable.

7.3.1 Switch Statements vs If Statements

To get an explanation for why `switch` statements are more efficient compared to `if` statements, we start by looking at the benchmarks. The main differences in the benchmarks is that the `if` benchmark needs to evaluate all of the `if` statements before getting to the one that is true, while the `switch` benchmark has cases. To figure out how significant this difference is, we look at the documentation.

In [111, p.96] we can see that "The target of br, br.s, and the conditional branches, is the address specified. The targets of switch are all of the addresses specified in the jump table." and in [111, p.392] we can see that "The switch instruction implements a jump table. ... The switch instruction pops value off the stack and compares it, as an unsigned integer, to n. If value is less than n, execution is transferred to the value'th target, where targets are numbered from 0 (i.e., a value of 0 takes the first target, a value of 1 takes the second target, and so on). If value is not less than n, execution continues at the next instruction (fall through).". This implies that when using a `switch` statement, unconditional jumps are utilized, which would be different than when using an `if` statement, where the statements are evaluated before the code is executed. This explains why `switch` statements are more efficient than `if` statements, as the switch condition is only evaluated once while in the `if` statements, it is evaluated every time we check if we are at the right code. To confirm this explanation, we move to the next step of result analysis and look at the IL code.

In the IL code it is clear to see there is a difference between the benchmarks. In the `switch` benchmark, a jump table is utilized to jump to the correct statement immediately. In the `if` benchmark all of the `if` statements before the wanted branch is evaluated, meaning this is the reason for the difference in performance.

The exact same behaviour can be seen for the comparison `switch` statement vs the equivalent `if` statement. For the full analysis see Section A.4.

7.3.2 Conditional Operator vs If Statements

The first step for looking at the differences between `conditional` operators and `if` statements is looking at the benchmarks. The only difference be-

tween the benchmarks is the statements used, where one uses `conditional` operators and the other uses `if` statements. This is not enough to explain the differences in performance, furthermore there is no information in the relevant documentation [117, 118] that explains these differences in performance, therefore we look at the IL code.

There are quite a few of differences in the IL code, which makes it surprising that the differences in performance is so small. The `if` benchmark has approximately double the amount of instructions despite being very close to the same performance as the `conditional` benchmark.

A cause of this could be that the `if` statement uses `cgt`, `ceq` and `brfalse.s` instructions instead of `ble.s`. This implies that comparing using greater than and equals is more efficient than comparing using less-than or equals and that the compiler makes this optimization automatically in some cases. To see if we can gain further insight, we look at the assembly code, which also shows that the compiler tries to avoid using the `jle` instruction. Further analysis of this is needed to find the reason for difference in performance, which we leave to future work in Chapter 10. The full analysis is seen in Section A.4.

7.3.3 If Statements vs If Else Statements vs If Else If Statements

The first step for looking at the differences between `if` statements, `if else` statements and `if else if` statements is looking at the benchmarks. There are no differences besides what we would expect, and there is no information in the relevant documentation [118] that explains the differences found in the results, therefore we look at the IL code.

Surprisingly, the IL code is completely equivalent, besides a single `nop` instruction that is not present in the `if` benchmark, so this gains no further insight into the differences in performance, therefore we look at the assembly code.

Once again, surprisingly, there is no difference between any of the benchmarks, besides that single `nop` instruction. Therefore we can conclude that there are some variances in our results that we have not been able to remove, which we will reflect over in Chapter 8. For the full analysis see Section A.4.

7.3.4 Summary

We find that `switch` statements are more efficient compared to `if` statements. This is because `switch` statements utilize jump tables after a single comparison instead of comparing in every statement to see if a block of code should be run. Furthermore, we find that there are differences between `conditional` operators and `if` statements, however finding conclusive evidence for the differences in performance is left to future work in Chapter 10. Besides this, we find that there is no reason for why there is a difference between the three `if` benchmarks, so the variances in the results found throughout the project are explored further in Chapter 8.

7.4 Collections

Next we look at the collections results, as these also had some interesting findings, which are:

- `Immutable List get` consumes over 40 times more energy than the other benchmarks in the `List Get` subgroup, except for `LinkedList`.
- `Linked List Removal` and `Immutable List Removal` consumes one third the energy of other list removal benchmarks.
- `Hashtable Get` and `Hashtable Get Random` consumes more than three times as much energy as `Dictionary Get` and `Dictionary Get Random`.
- `Read Only Dictionary Get` consumes more energy than `Dictionary Get`.

We start by examining the reason behind `Immutable List get` consuming 40 times more energy than other members, except for `LinkedList`, in the `List Get` subgroup.

7.4.1 Immutable List Get

The first step in analyzing why using `Immutable List get` consumes more energy than all other benchmarks in the `List Get` subgroup except for `LinkedList`, is to look at the benchmark and compare it with the others. Comparing the code from the `Immutablelist` and `ListGet` benchmarks we see that the only differences are ones we would expect, i.e. that the

ImmutableListGet benchmarks uses an Immutablelist and the ListGet benchmark uses a List. We have not found anything that explains why there is this difference in performance in the documentation [119], therefore we look at the the IL code, as the next step of the process.

The IL is identical, except for the type of list used, therefore we look at the assembly code.

The assembly does not give any clear answer to why ListGet is more efficient then ImmutableListGet, as ListGet contains more instructions and an additional call compared to ImmutableListGet. Thus, knowledge of the C# compiler is needed to gain further insight into these differences, meaning we leave that for future work in Chapter 10. The full analysis is seen in Section A.4.

7.4.2 Linked and Immutable List Removal

The first step in analyzing why LinkedList and ImmutableList consumes one-third the energy of other list removal benchmarks, is to compare it to other benchmarks in the subgroup. Looking at the benchmarks we encounter differences in how elements are added and removed from the list. Looking at the these differences we find that LinkedList's RemoveAt method is constant according to [120], which could explain why LinkedList has lower energy consumption. However explanation for ImmutableList is not found in the documentation [119]. As such we move onto step three of result analysis and look at the IL code.

While the IL code showed differences between LinkedList, ImmutableList and List, these differences could not explain the lower energy consumption, meaning we look at the assembly code.

In the assembly code we unexpectedly find that LinkedList's Remove method and ImmutableList's RemoveAt have more instructions than List's RemoveAt. However the results have show that LinkedList and ImmutableList are more efficient, therefore this requires further analysis, which we leave for future work in Chapter 10. The full analysis conducted to reach these results can be seen in Section A.4.

7.4.3 Hashtable Get & Get random

The first step for analyzing why the performance of the Get method on a Hashtable consumes more energy than the Get method on a Dictionary, is to look at the benchmarks. Looking at the benchmarks we encounter one

difference, which is that the `Hashtable` benchmarks needs to cast a value to an `int`, specifically unboxing an object to accomplish the same work as a `Dictionary`. Given this we look at the documentation for unboxing, and in [121] it is stated that unboxing is a computationally expensive process, and doing unboxing can lead to casting taking four times as long. This casting is not needed for `Dictionary` and as such we conclude that this casting and subsequent unboxing is the reason behind the increased energy consumption. The analysis conducted to reach this conclusion is seen in Section A.4.

7.4.4 Read Only Dictionary Get vs Dictionary Get

The first step for examining why the performance of the `Get` method on a `ReadOnlyDictionary` is less efficient than the `Get` method on a `Dictionary` is looking at the benchmarks. There are no differences between these two benchmarks that are unexpected, and there is no information in the documentation [122, 123] that explains this difference either, therefore we look at the IL code.

The IL code also shows that there are no unexpected differences between the benchmarks, meaning we look at the assembly code.

In the assembly code, we unexpectedly find that the `Get` method on the `Dictionary` has more instructions, including an extra `call` instruction which we would expect to make it less efficient, however the results show that using the `Get` method on a `Dictionary` is more efficient, therefore this requires further analysis which we leave for future work in Chapter 10. The analysis conducted to reach these conclusions can be seen in Section A.4.

7.4.5 Summary

In this section we find that knowledge of the C# compiler is needed to figure out why the `get` operation on an `Immutable List` is less efficient than `get` operations on other types of lists, therefore we leave this for future work in Chapter 10.

Besides this, we find that removing elements from a `LinkedList` is of constant complexity operation according to [120], however the complexity of removing from an `ImmutableList` is not present in the documentation. With regards to the `ImmutableList`, we can not get any insight into why this is more efficient just by looking at the IL code and assembly code, therefore

we find that knowledge of the C# compiler is needed and we, therefore, leave that for future work in Chapter 10.

We find that in our benchmarks for using the `Get` operation on a `Hashtable` we perform a cast to an `int`, which according to [121] is computationally expensive, and it can lead to being four times as inefficient. This cast is not needed for `Dictionary` and we can therefore conclude that this is why there is a difference in performance between these two constructs.

Lastly, we find that knowledge of the C# compiler is needed to figure out why the `get` operation on a `ReadOnlyDictionary` is less efficient than the `get` operation on a `Dictionary`, as the only difference between these two is that `Dictionary` has more instructions in the assembly code, which is opposite to what we would expect, therefore we leave further analysis of this to future work in Chapter 10.

7.5 String Concatenation

We examine the string concatenation results, as these had the following findings:

- Approaches of concatenating strings vary in efficiency and speed for approaches that appear similar, like `string.format` and `string` interpolation, and
- `StringBuilder` is significantly more efficient and faster than all other approaches.

We start by examining why there are differences in the approaches that look similar.

7.5.1 Similar Approaches

The first step for examining the differences between similar approaches to `string` concatenation is looking at the benchmarks. In this case we look at the `string.Format` benchmark and the `string` interpolation benchmark, as these approaches are similar but have a large difference in results.

In the benchmarks there are no differences besides what we would expect, therefore we look at the documentation. In [124] it is written that `string` interpolation is recommended over `string.Format` because it is

more flexible and readable, however this does not explain why `string` interpolation is more efficient. In [125] it is also written that `string` interpolation is more readable than `string.Format`. No direct explanation for why there is a difference between these two can be found in the documentation, meaning we look at the IL code.

Here we find that `string.Format` uses an array to keep track of where the different objects should be placed in the `string` while `string` interpolation uses a built-in type called `DefaultInterpolatedStringHandler`. To see if we can gain further insight, we look into the assembly code. From the assembly code we see that built-in methods are called which is the main difference between the benchmarks, which would require knowledge of the C# compiler to understand why this effects the performance in the way it does, therefore we leave this to future work in Chapter 10. The full analysis is seen in Section A.4.

7.5.2 StringBuilder

To figure out why `StringBuilder` is the most efficient method of concatenating `strings`, we examine the benchmark compared to the second most efficient method of concatenating (non-constant) string, in this case `string` interpolation.

Here we do not find any unexpected differences, furthermore, there is not any information in the relevant documentation [126] that explains the difference in performance, therefore we go straight to the IL code.

In the IL code, we can see that the main differences is that several calls are made to methods that handle `StringBuilder` and `string` interpolation. In fact, the calls to `StringBuilder` are `virtcall` instructions, which intuition would consider more expensive as you would have to check if the method has been overridden, however this is not the case. This means the difference is in how a `StringBuilder` is implemented by the compiler compared to how `string` interpolation is implemented.

To see if we can gain further insight, we look into the assembly code. From the assembly code we see that built-in methods are called which is likely to be the cause of the performance differences between the benchmarks, which would require knowledge of the C# compiler to understand why this affects the performance in the way it does, therefore we leave this to future work in Chapter 10. The full analysis of `StringBuilder` can be seen in Section A.4.

7.5.3 Summary

In this section, we find that even though some approaches to concatenation may look similar, they behave differently when compiled, and therefore there are differences in efficiency between these. Besides this, we find that `StringBuilder` has a more efficient implementation of concatenating in the compiler, making this the most efficient choice when concatenating.

Finding further explanation for these results would require knowledge of the C# compiler, which we leave to future work in Chapter 10.

7.6 Invocation

The next results we analyse is the invocation results, as these had the following findings:

- Time and package energy consumption does not seem to correlate for `Local Function Invocation`, this is interesting to us, as it is contrary to the rest of our results.
- Delegate using reflection is faster and consumes less energy than just using reflection, using reflection with flags or using static reflection.

We start by examining why time and package energy consumption does not correlate for `Local Function Invocation`.

7.6.1 Local Function vs Local Function Invocation

The first step for examining why the correlation between Elapsed Time and package energy does not hold for `LocalFunctionInvocation` but it holds for all the other local function benchmarks is checking the benchmarks. In this case we look at the four relevant benchmarks: `LocalFunction`, `LocalFunctionInvocation`, `LocalStaticFunction` and `LocalStaticFunctionInvocation`.

In the benchmarks, there are no differences that we would not expect, furthermore there is no information in the relevant documentation [127] that would explain the results we have found, therefore we move onto step three of the process and look at the IL code.

In the IL code it is not possible to find the reason for why the correlation does not hold for `LocalFunctionInvocation`, therefore we examine the assembly code. The assembly code gives the same picture as the IL code,

therefore we can not find the reason for why the correlation does not hold and we therefore leave that for future work in Chapter 10. For a look at the full analysis look at Section A.4

7.6.2 Reflection

The first step for looking at why delegate using reflection is more efficient compared to other types of reflection, is to examine the benchmarks for `ReflectionDelegate` and `ReflectionFlags` as these are similar but `ReflectionFlags` does not use delegate.

In the benchmarks there are no differences besides the language construct used which is expected, therefore we look at the documentation. According to [68] "Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.", which implies that pointers are used behind the scenes, which would make this more efficient than the alternatives.

Because of this, it is possible that the reflection with delegates only needs to find the method once instead of every time the method is called. To get further insight, we examine the IL code for these benchmarks.

In the IL code we can observe that the main differences are in the built-in methods in the compiler, therefore this does not help gain further insight into why the difference in performance exists between these two benchmarks. We therefore examine the assembly code to see if we can get further insight.

In the assembly code we can see when using the delegate, no built-in method calls are done and there are significantly fewer instructions done, which explains why the difference in performance is so large. In Section A.4 the full process for drawing this conclusion is seen.

7.6.3 Summary

In this section we were unable to find the reason for why time and package energy do not correlate directly for `Local Function Invocation` and `Local Function` compared to `Local Static Function Invocation` and `Local Static Function`, therefore we leave this for future work in Chapter 10.

Furthermore, we find that using `reflection` with `delegates` is more efficient than other types of `reflection` because using `reflection` with

`delegates` enables the compiler to optimize further when creating the assembly code.

7.7 Objects

The next results we analyse is the objects results, as these had the following finding:

- `Class Method` has a higher elapsed time than `Class Method Static`, while it consumes less energy.

We examine why time and package energy consumption does not correlate for `Class Method` and `Class Method Static`.

7.7.1 Class Method vs Class Method Static

The first step for examining why the correlation between elapsed eime and eackage energy does not hold for `Class Method` and `Class Method Static` is looking at the benchmarks.

In the benchmarks we do not find any relevant differences that would explain this, and there is no known documentation that helps explain these differences in performance, therefore we look at the IL code.

When looking at the IL code we can see that there are no new differences that did not exist in the C# code, therefore we go to the assembly code.

In the assembly code we can see that there is a difference between using `instance` fields versus `static` fields. These differences are mainly that there are some extra `call` instructions when using `static` fields compared to `instance` fields. These differences could explain why there is a correlation mismatch between the two benchmarks, however more analysis is needed to make sure this is the case, which we leave for future work in Chapter 10. The analysis conducted to reach these conclusions can be seen in Section A.4

7.7.2 Summary

In this section we were unable to find conclusive evidence for why time and package energy does not correlate directly for `Class Method` and `Class Method Static`, however, there seems to be a difference in the assembly

code when using `instance` fields versus `static` fields which could be the reason behind the correlation mismatch, however, more research is needed which we leave for future work in Chapter 10.

7.8 Inheritance

We look at the inheritance results, as these had the following findings:

- `Inheritance virtual` is more efficient compared to other types of inheritance, and
- `No Inheritance` is the least efficient.

We start by looking into why `Inheritance Virtual` is more efficient compared to other types of inheritance.

7.8.1 Inheritance Virtual

The first step for examining why the performance of `Inheritance Virtual` is more efficient than other types of inheritance is looking at benchmarks. Here we decide to look at the `Inheritance` benchmark, to see if we can find a difference between the two. When looking at the benchmarks themselves, no difference besides what is expected is found, therefore documentation is looked at.

In [128] it is stated that "When a virtual method is invoked, the run-time type of the object is checked for an overriding member." which implies that a check is done when a `virtual` method is called, this does not explain why calling a `virtual` method is more efficient compared to an inherited method, in fact, it would imply the opposite making this result unexpected.

Because the documentation did not provide an explanation, we look at the IL code. The IL code for both of these benchmarks is equivalent making the results even more unexpected, we therefore look at the assembly code to see if they stay equal or they diverge.

In the assembly code, there is a difference, where the `Inheritance` benchmark uses a `call` instruction on a `CLRStub...` method, which previous experiments have shown is less efficient than calling a pointer, therefore we conclude that this is why `Inheritance Virtual` is more efficient than `Inheritance`, however, further analysis is needed to figure out the specifics of why this is the case. An interesting finding here is that even though the

IL code was equivalent between the two benchmarks, the assembly code is not. The analysis conducted to reach these conclusions can be seen in Section A.4.

7.8.2 No Inheritance

The first step for examining why the performance of `No Inheritance` is less efficient than any type of inheritance is looking at benchmarks. Here we decide to look at the `Inheritance` benchmark, to see if we can find a difference between the two. When looking at the benchmarks themselves, no difference besides what is expected is found, therefore documentation is looked at.

The documentation [79] does not provide an explanation for the results, therefore we look at the IL code. The IL code for both of these benchmarks is equivalent making the results even more unexpected, we therefore look at the assembly code to see if they stay equal or diverge.

In the assembly code, there is a difference, where the `No Inheritance` benchmark uses a `call` instruction on a `CLRStub...` method, while the `Inheritance` benchmark uses a `call` instruction on a `method stub...`. We therefore conclude that this is why `Inheritance` is more efficient than `No Inheritance`, however, further analysis is needed to figure out the specifics of why this is the case. An interesting finding here is that even though the IL code was equivalent between the two benchmarks, the assembly code is not. Furthermore, it is very surprising that the assembly code for the `Inheritance` benchmark has changed when running the `Inheritance` and `No Inheritance` benchmarks compared to when running the `Inheritance` and `Inheritance Virtual` benchmarks. More analysis into the `Inheritance` results are left for future work in Chapter 10. The analysis conducted to reach these conclusions can be seen in Section A.4.

7.8.3 Summary

We find that calling a virtual method directly is more efficient than calling a method via inheritance. This is because the `call` instruction in assembly code calls `CLRStub...` instead of directly calling the method when using inheritance.

Furthermore, we find that calling a method directly in C#, causes the compiler to generate assembly code that calls `CLRStub...`, which makes this less efficient than inheritance. Importantly, when running the `Inheritance`

benchmark together with the `No Inheritance` benchmark instead of the `Inheritance Virtual` benchmark, the generated assembly code calls `Method stub...` instead of `CLRStub...` making this more efficient than `No Inheritance`.

Further analysis of the `Inheritance` results are left for future work in Chapter 10.

7.9 Exceptions

We look at the exceptions results, as these had the following findings:

- Using a `try catch` statement when `exceptions` are thrown 50% of the time is 440 times slower than checking for the `exception` condition using an `if` statement,
- Throwing an `ArgumentException` uses about twice as much energy as causing a `DivideByZeroException`, and
- Wrapping code in a `try catch` statement increases Package Energy consumption with approximately 110% when no `exceptions` are thrown.

We start by looking into why using a `try catch` statement is 440 times slower than checking for the `exception` condition using an `if` statement.

7.9.1 Try Catch vs If

The first step for looking at why there is a difference between catching an `exception` and checking if the `exception` would be thrown with an `if` statement, is checking the benchmarks.

In the benchmarks we find no differences besides the one we would expect, therefore we look at the documentation.

In [129] it is stated that "When an exception occurs, the system searches for the nearest catch clause that can handle the exception, as determined by the run-time type of the exception." and in [130] it is stated that "When an exception is thrown, the common language runtime (CLR) looks for the catch statement that handles this exception. If the currently executing method does not contain such a catch block, the CLR looks at the method that called the current method, and so on up the call stack." meaning that when an `exception` occurs, the call stack is looked through to find a way to handle the `exception`. It makes sense that this is significantly more

expensive than executing an `if` statement, therefore we can conclude that `exceptions` need to look through the call stack, which is more expensive than executing an `if` statement.

7.9.2 ArgumentException vs DivideByZeroException

The first step for looking at why there is a difference between throwing an `ArgumentException` and throwing a `DivideByZeroException`, is checking the benchmarks.

The main difference is that in the `ArgumentException` benchmark, we create an `ArgumentException` object to throw, while in the `DivideByZeroException` benchmark, we implicitly throw by dividing by zero. Furthermore, more operations are done in the `DivideByZeroException` benchmark, compared to the `ArgumentException` benchmark, which would imply `DivideByZeroException` should perform worse, however the opposite is the case. Therefore we look at the documentation for each of these `exceptions`.

In [131] we can see that a lot of `exceptions` are derived from `ArgumentException`, which could imply that a lookup for which one is thrown is done, while in [132] we can see that no `exceptions` are derived from `DivideByZeroException`. Besides this, there is no explanation in the documentation for the differences in these exceptions, therefore we look at the IL code.

The IL code gives the same picture as the C# code, therefore we go straight to the assembly code. In the assembly code for the `ArgumentException` benchmark, there are some `call` instructions, while there are no `call` instructions in the `DivideByZeroException` benchmark. Furthermore, the `catch` blocks are not generated until the exception is thrown. When stepping through the code, we can not continue after a specific instruction, `call CLRStub...` in the `ArgumentException` benchmark, however this is not present in the `DivideByZeroException` benchmark, meaning this could be where the difference in performance comes from. However, we can not conclude anything for certain here as it would require knowledge of the C# compiler so we leave that for future work in Chapter 10.

7.9.3 Try Catch with No Exception

The first step for looking at why there is a difference in performance between having a `try-catch` block without `exceptions` ever thrown and not having a `try-catch` block, is looking at the benchmarks.

When looking at the benchmarks, we find no difference besides what we would expect, therefore we look at the documentation. In [130] it is stated that "The try block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully." implying that `try` blocks have something extra that would not exist if the `try` block was not there. To figure out the differences, we look at the IL code.

The only differences here is that there is a `try` block around the operations, and a `leave.s` instruction at the end of the `try` block, in the `try-catch` benchmark. Furthermore, there is a `catch` block, however this is never reached so not relevant. We look at the assembly code to gain insight into how this is reflected there.

Surprisingly, in the assembly code we find that there is no difference between the two benchmarks, besides a single `nop` instruction in the `try-catch` benchmark. Therefore we can not conclude why there is a difference in performance between these two benchmarks, and we leave further exploration of this to future work in Chapter 10.

7.9.4 Summary

We find that using an `if` statement is less expensive than using a `try catch` statement because `try catch` statements need to look through the call stack for the `catch` statement. Furthermore, we find that when implicitly throwing a `DivideByZeroException`, we immediately go to the `catch` block, while when explicitly throwing an `ArgumentException`, a `call` instruction is created that has some impact in performance. Further exploration of that is needed to get an explanation for the difference in performance. Besides this, we can not find an explanation for why there is a difference between having a `try-catch` block around code that never throws an exception versus not having a `try-catch` block around that code. This means further exploration of this concept is needed to get an explanation for the performance difference which we leave to future work in Chapter 10.

Chapter 8

Reflections

In this chapter we reflect over our project. We reflect over our type of benchmarks, i.e. if we should have created macrobenchmarks instead of microbenchmarks or if it was the right choice. We reflect over variances we have seen in the results, and over an issue we had when creating a benchmark for recursion. Furthermore we reflect over our choice not to turn off compiler optimizations. We then end the chapter by reflecting on our work process.

8.1 Choice of Benchmark Type

In Section 4.3 we elected to use microbenchmarks, this choice was done to get insight into the specifics of the different language constructs without having irrelevant code around it, however, this does not necessarily have the same effect as when the language constructs are part of a larger codebase.

We think the choice of microbenchmarks has given insight into what the effect of individual language constructs are and made it possible to test a larger selection of language constructs, as choosing macrobenchmarks would require more time to create the same number of experiments. However having macrobenchmarks would be a good way to ensure that the results are useful in a larger context.

8.2 Result Variance

Throughout the project, we have observed that benchmark results can vary by approximately 1% between each run of individual benchmarks, an ex-

ample of which is shown in Section A.5. This means that if we observe that two benchmarks are within 2 percent of each other, they are too close to say for sure if one is more efficient than the other, as one benchmark may have an efficient run while the other benchmark has an inefficient run. The variance is to such an extent that the same benchmark can have a significant difference (p-value less than 0,05) from itself, meaning calculating the p-value is not enough to correct this issue. We have observed that the differences may occur in the same run of the full benchmark suite, so one benchmark may have an efficient run while another may have an inefficient run. An example could be that `nint` shows that it is 1,5 percent less efficient than `long`, it may show in another run that `nint` is 0,5 percent more efficient than `long`. While observed we do not know exactly why they exhibit such a behavior. We suspect that it is caused by an external variable that we have not taken into account, most likely being some part of the computer setup, that we do not recognize as potentially influencing our results. We acknowledge the variety might be a side effect of using microbenchmarks. The reasoning behind this thought is that very small code segments are used for benchmarking and because of this even small fluctuations appear to have a somewhat large effect. As mentioned earlier this might not be the case if we were to have used macrobenchmarks. This means that our results vary when it comes to their energy consumption and elapsed time, and as such the degree to which a certain language construct is better than another can vary from run to run.

8.3 Recursion Issues

While most of the language constructs have been tested, there are however issues in the Loops group from Table 3.1. The problem is recursion. Testing recursion is not possible using our approach. The specific issue lies in the number of iterations needed to reach the 0,25 seconds specified in Section 4.5. Going beyond 174.601 iterations results in a stack buffer overflow which is not a sufficient number of iterations. Because of this, there are no results regarding recursion and it thus cannot be compared to other constructs within the group. Workarounds exist, but they result in altering the benchmark behavior to a degree that it is no longer comparable to other loop constructs. Workarounds could be to run a set amount of recursions at a time, enough times to compare to other benchmarks, however, this would include other types of loops and therefore make the comparisons dubious.

8.4 Uncontrollable Compiler Optimizations

We have elected not to turn off optimizations. We assume the results reflect how the language constructs affect energy consumption in real-life scenarios, and optimization would be enabled in such scenarios. Part of the process of writing our benchmarks has been to ensure that they are not optimized away by the compiler. We suspect that this has not always been successful and that results for some of the language constructs have been skewed by this. Among our results, we assume that Field Mutability, Object, and Bitwise Operators are influenced by uncontrollable compiler optimizations which question the validity of the results for these benchmarks.

8.5 Use of Libraries

Building upon [9], using [107] and [133] has helped save time in the development of the framework, making it possible to use the time on creating a larger benchmark suite and analysing why the results occurred in the way they did.

8.6 Work Process

Overall we consider our work process to have worked well. Utilizing a hybrid of the plan-driven and agile processes has worked well for us, as following a loose plan has made it easier to keep track of how well we manage our time, but also being agile when unforeseen challenges occur is useful so we can adapt our plan quickly.

Having daily stand-up meetings to keep up with each other has worked well in keeping an overview of how far we were at different stages of the project, as well as keeping an overview of what we were all doing. Using a kanban board in the form of GitKraken Boards has helped in keeping an overview of what stage we are in the project and have helped keep an overview of what everyone is doing so we can spread the work-load.

Lastly, using GitLab has worked well to keep track of the framework and benchmarks we have produced during the project, here we decided to create branches for each new feature and fix. When merging these branches with the main branch, we created merge requests in which at least two project members have to approve the merge request before the code could

be merged. These two project members need to have been different from the ones who have written the code in the merge request. Furthermore, we have created tests that are run automatically via continuous integration every time any changes are pushed to any branch. All the tests are required to pass before a merge request could be accepted. The merge requests needing to be approved and the tests needing to pass have helped improve the quality of our code.

Chapter 9

Conclusion

We have created a framework that can be used to measure the energy consumption of benchmarks. We have used the framework to measure the energy efficiency of language constructs by creating microbenchmarks focused on testing the individual language constructs. In the span of this project we have created a total of 316 benchmarks. The results of these benchmarks can be used to provide a basis for developing guidelines and tools which can help facilitate better energy utilization and energy awareness for programming in the C# language. Furthermore, this research provides groundwork for further research into C# regarding energy awareness. As the last step, we have created a process for analysing the results of the language constructs which attempts to explain the chosen benchmark results. Through the process, we have achieved explanations for multiple benchmark subgroups.

Overall, we have four contributions following this project:

1. A framework for measuring benchmarks with regards to energy consumption,
2. a microbenchmark suite which measures a large selection of language constructs,
3. the results of running these microbenchmarks with our framework which helps developers choose what language construct to use, and
4. the reason behind some of these results using the process we have created for analysing results.

All of the results can be found in Section A.3. The most interesting or unexpected results from running the benchmarks with our framework are as follows:

- Unsigned integer datatypes are at least 1% more efficient than signed integer datatypes, except
- `ushort` is only 1% more efficient than `short`,
- `uint` is the most efficient datatype,
- `switch` is more efficient than `if` statements when applicable,
- Conditional operators are more efficient than `if` statements when looking at package energy,
- `if` consumes the least amount of energy while being the slowest of the benchmarks it is compared to,
- `Immutable List get` consumes over 40 times more energy than the other benchmarks in the `List Get` subgroup, except for `LinkedList`,
- `Linked List Removal` and `Immutable List Removal` consumes one third the energy of other list removal benchmarks,
- `Hashtable Get` and `Hashtable Get Random` consumes more than three times as much energy as `Dictionary Get` and `Dictionary Get Random`,
- `Read Only Dictionary Get` consumes more energy than `Dictionary Get`,
- Approaches of concatenating strings vary in efficiency and speed for approaches that look similar, like `string.Format` and string interpolation,
- `StringBuilder` is significantly more efficient and faster than all other approaches,
- Time and package energy consumption does not seem to correlate for `Local Function Invocation`, this is interesting to us, as it is contrary to most of our results,
- Delegate using reflection is faster and consumes less energy than just using reflection, using reflection with flags or using static reflection,

- `Class Method` has a higher elapsed time than `Class Method Static`, while it consumes less energy,
- `Inheritance virtual` is more efficient compared to other types of inheritance,
- no inheritance is the least efficient,
- Using a `try catch` statement when the `DivideByZeroException` is thrown 50% of the time, is 440 times slower than checking for the `exception` condition using an if statement,
- Throwing an `ArgumentException` uses about twice as much energy as causing a `DivideByZeroException`, and
- Wrapping code in a `try catch` statement increases package energy consumption with approximately 110% when no `exceptions` are thrown.

Throughout the project we have learned that there are a lot of variables when running benchmarks on a computer, some of which we have not been able to correct for, meaning there is an error of up to 1% for each run of individual benchmarks.

Besides this, we have learned that writing microbenchmarks is not trivial as the C# compiler often is good at optimizing code away. However, other times the C# compiler is not good at optimizing code away, meaning that a benchmark that tests e.g. addition can not automatically be used when testing subtraction where the only difference is that one uses addition and the other subtraction. This means, whenever a benchmark is created and have been confirmed not to be optimized away by the C# compiler, it does not mean that the benchmark can be used for the entire category.

To end the conclusion, we conclude upon the problem statement with accompanying research questions, which is as follows:

What are the energy consumption effects of different language constructs in C#?

1. *How can the energy efficiency of language constructs be measured?*
2. *How can a suite of experiments/benchmarks be made to measure the energy efficiency of language constructs?*
3. *Why is the energy consumption different between similar language constructs?*

The implementation of our framework, which is used to measure the energy efficiency of language constructs can be seen in Chapter 5, while the full explanation of the framework implementation can be seen in Section A.2. The energy consumption effects of different language constructs in C# have been found and the most interesting of these results are presented in Chapter 6. All of the results can be found in Section A.3. The analysis of these results can be seen in Chapter 7 and Section A.4.

Chapter 10

Future Work

This chapter presents future work. We present ideas for both extending the research we have conducted and creating tools that developers can use to improve the energy consumption of their programs.

Information Tools

One use of our research could be to create informative tools which help developers be energy aware when programming. An example of such a tool could be an IDE extension that informs the user of alternative language constructs which can impact the energy consumption of their code positively. Examples of tools that have been created to improve energy efficiency in software is seen in [49], [50] and [51]. It could also be a wiki in which a developer can look up which language constructs are useful when improving energy consumption.

Other Programming Languages

Additionally, an interesting extension of the research would be to create and perform identical benchmarks using the Java language, to compare the C# and Java language constructs. This could provide information on whether the behavior of language constructs is significantly different or the same energy consumption is to be expected across a spectrum of languages, such as interpreted, managed, and compiled languages or in different paradigms, such as imperative, object-oriented or functional programming. Besides this, other programming languages could be researched as well, to see how other languages' energy efficiency compares to C# and Java. According

to [12] the most popular programming languages are Python, C, Java, C++, and C#, therefore these would be clear candidates for other programming languages.

Examples of this being done in Java is seen in [4], and [11]. Examples of comparison of paradigms can be seen in [9].

Result Analysis

As we analysed the different benchmark results we discovered results that could not be properly explained using our analysis method.

We discovered that `StringBuilder` was a more efficient implementation of concatenating, which makes it the most efficient choice for concatenation. Due to the large differences in the compiled code, we are not able to discern what specific difference that makes the `StringBuilder` more efficient. As such it would be interesting to further investigate the compiler to find an explanation as to why the `StringBuilder` is better.

Investigating the C# compiler is a large task as the codebase is both large and complex, however, it is possible as the source code is available [134]. In addition to this we consider the .Net Just-In-Time (JIT) compiler as an important piece to understand. The investigation of the .Net JIT compiler is also a large task for the same reasons. Besides this, knowledge of the runtime [135] might be necessary to explain the results, as the compiler alone may not have enough information to explain the results seen.

Additionally we discovered interesting results regarding `ushort` and `short` as well as `byte` and `sbyte`. By looking at the assembly code we found one difference between the `short` and `byte` being the use of `movsx` and `movzx` instructions. We consider that this difference does not explain the differing results of `short` and `byte`. Therefore it would be interesting to further investigate the compiler to find an explanation as to why this difference exists.

In our results, we have observed that the results for inheritance change significantly between runs. It is to such a degree that ranking which language construct consumes the least energy changes between runs, as can be seen in Section A.6. If we look at the result analysis in Section A.4, we see that the same benchmark creates two different sets of assembly code depending on what code was run before it. The difference in assembly code could have an influence on the results. However, we are not sure what causes these benchmarks to vary to such a degree. As such, it would be interesting to further investigate what happens in the C# compiler to make

these differences occur.

Benchmark Metrics

During the analysis process, we focus on the package energy and elapsed time of the benchmarks. In addition to these two measurements, we also measure the DRAM energy consumption, where we in the span of this project have chosen not to further examine the DRAM energy consumption of the benchmarks. It would therefore be an interesting addition to further investigate the behavior of C# language constructs in regards to their DRAM energy usage.

Additional Benchmarks

During the span of the project, we have gained a better understanding of the language constructs of C#. We have discovered that there are several variations and cases within C# which allow for further testing. Additional benchmarks can be created to further the research of this project, examples of what such benchmarks could do are:

- test the relationship between constructed strings and constant strings,
- look at the energy cost of local functions using variables from enclosing scopes,
- see how using lambda functions affect energy consumption,
- test parameter passing using `in` or `ref`, when using value types or reference types, and
- compare the energy consumption of using the `out` parameter versus using a tuple return type.

Besides these benchmarks, creating and utilizing macrobenchmarks would allow further insight into how energy consumption is impacted by multiple uses of a language construct, the results generated from these would also likely be less prone to influence from external variables. Using macrobenchmarks would also allow for testing combinations of language constructs.

Utilizing applications and refactoring them according to the data collected, would allow for insight into how the language construct functions in non-synthetic situations. Doing this would not be trivial, as there may

be cases where changing language constructs may alter the behavior of an application, and as such the application from before and after refactoring would not be comparable.

Lastly, creating application-wide benchmarks would be interesting, as a lot of applications exist which utilize multiple languages. We would therefore not be able to fully test these types of applications by looking at a single language.

Unrolling Loops and Outliers

We have found that unrolling a loop causes a significant loss in efficiency, as seen in Section A.1.1. Besides this we also found that the first run of a benchmark is an outlier compared to the rest of the runs of a benchmark. Researching the cause of these results could be interesting, and we hypothesize that an area of research could be within the JIT compiler as this is likely to be part of the cause for these results.

10.1 Related Work Areas

In Section 2.2 we looked at several research areas and papers. Some of these are of interest for further research. Extending the research on language constructs in these areas would allow developers to better understand the implications of using different language constructs and thereby allow for reasoning when one construct is better for the situation. Two such areas are the relation between execution-time and energy consumption, and maintainability.

Execution-time

In regards to the relation between energy consumption and execution-time, more research into this topic is needed. We have concerned ourselves primarily with the energy consumption of language constructs. In our experience, there are cases where a developer is not willing or able to accept that a program will have a longer execution-time as a side effect of reducing energy consumption. As such it makes sense to extend the research and look at the execution/elapsed time. As part of this looking at the relationship between time and energy would allow developers to better choose a

compromise between energy and time efficiency, depending on the situation. The relationship between execution time and energy efficiency has been explored in [39].

Maintainability

One area well suited for further work with language constructs is maintainability. We have already seen examples of this research area from the papers [43], [44] and [45]. We have looked at energy consumption, but maintainability is important for maintaining an application across its lifetime. As such it could be relevant to look at how refactoring language constructs from the different groups impact maintainability and to which degree, here the approach presented in [43] could be used. This could then be built upon further by looking at combinations of the different groups and seeing how those different combinations impact maintainability as seen in [45]. If choosing to combine groups there should also be research into how the combinations impact energy consumption, an example of a paper that looks at this area is [44], which looks at refactorings in isolation. There could even be research into what maintainability concerns are impacted the most, by both individual and combined changes.

Bibliography

[1] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. "Recalibrating global data center energy-use estimates". In: *Science* 367.6481 (2020), pp. 984–986.

[2] Nicola Jones. "How to stop data centres from gobbling up the world's electricity". In: *Nature* 561.7722 (2018), pp. 163–167.

[3] Luís Gabriel Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, Gilberto Melfe, and João Paulo Fernandes. "Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 517–528. DOI: 10.1109/SANER.2016.85.

[4] Mohit Kumar, Youhuizi Li, and Weisong Shi. "Energy consumption in Java: An early experience". In: *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*. 2017, pp. 1–8. DOI: 10.1109/IGCC.2017.8323579.

[5] Christian Bunse, Hagen Höpfner, Essam Mansour, and Suman Roychoudhury. "Exploring the energy consumption of data sorting algorithms in embedded and mobile environments". In: *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*. IEEE. 2009, pp. 600–607.

[6] Hesham Hassan, Ahmed Moussa, and Ibrahim Farag. "Performance vs. Power and Energy Consumption: Impact of Coding Style and Compiler". In: *International Journal of Advanced Computer Science and Applications* 8 (2017-12). DOI: 10.14569/IJACSA.2017.081217.

[7] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. "Energy efficiency across programming languages: how do energy, time, and memory relate?"

In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. 2017, pp. 256–267.

[8] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. "Towards a Green Ranking for Programming Languages". In: *Proceedings of the 21st Brazilian Symposium on Programming Languages*. SBLP 2017. Fortaleza, CE, Brazil: Association for Computing Machinery, 2017. ISBN: 9781450353892. DOI: 10.1145/3125374.3125382. URL: https://dl.acm.org/doi/10.1145/3125374.3125382.

[9] Lars Rechter, Martin Jensen, Jacob Ruberg Nørhave, Casper Susgaard Nielsen, and Anne Benedicte Abildgaard Ejsing. *The Influence of Programming Paradigms on Energy Consumption*. Aalborg University, 2021.

[10] Gustavo Pinto and Fernando Castor. "Energy Efficiency: A New Concern for Application Software Developers". In: *Commun. ACM* 60.12 (2017-11), pp. 68–75. ISSN: 0001-0782. DOI: 10.1145/3154384. URL: https://doi.org/10.1145/3154384.

[11] Mohit Kumar. *Improving Energy Consumption of Java Programs*. Wayne State University, 2019.

[12] TIOBE. *TIOBE Index for August 2021*. 2021-08. URL: https://www.tiobe.com/tiobe-index/ (visited on 2021-09-08).

[13] Ian Sommerville. *Software engineering*. Harlow Singapore: Pearson, 2016. ISBN: 1-292-09613-6.

[14] LLC. Axosoft. *GitKraken Boards*. 2021. URL: https://www.gitkraken.com/boards (visited on 2021-10-01).

[15] Daniel Hackenberg, Thomas Ilsche, Robert Schöne, Daniel Molka, Maik Schmidt, and Wolfgang E. Nagel. "Power measurement techniques on standard compute nodes: A quantitative comparison". In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2013, pp. 194–204. DOI: 10.1109/ISPASS.2013.6557170.

[16] Girish Bekaroo, Chandradeo Bokhoree, and Colin Pattinson. "Power Measurement of Computers: Analysis of the Effectiveness of the Software Based Approach". In: *International Journal of Emerging Technology and Advanced Engineering* 4 (2014-05), pp. 755–762.

[17] Vernier. *Why did you discontinue Watts Up Pro? Are there alternatives?* 2020-08. URL: https://www.vernier.com/til/3763 (visited on 2021-10-11).

[18] Jason M Hirst, Jonathan R Miller, Brent A Kaplan, and Derek D Reed. "Watts Up? Pro AC Power Meter for Automated Energy Recording". In: *Behavior Analysis in Practice* 6.1 (2013), pp. 82–95.

[19] P3 INTERNATIONAL. *Kill A Watt™ EZ Operation Manual P4460*. 2006. URL: https://www.mnpower.com/ProgramsRebates/KillAWattEZOperationManual (visited on 2021-09-29).

[20] Justin Cooper. *Tweet-a-watt*. 2014. URL: https://learn.adafruit.com/tweet-a-watt/ (visited on 2021-09-29).

[21] SPEC. *The PTDaemon Tool*. URL: https://www.spec.org/power/docs/SPECpower-Device_List.html#mozTocId266535 (visited on 2021-10-04).

[22] SPEC. *Power Analyzer Acceptance Process*. URL: https://www.spec.org/power/docs/SPEC-Power_Analyzer_Acceptance_Process.pdf (visited on 2021-10-04).

[23] Jóakim von Kistowski, Klaus-Dieter Lange, Jeremy A. Arnold, Sanjay Sharma, Johann Pais, and Hansfried Block. "Measuring and Benchmarking Power Consumption and Energy Efficiency". In: ICPE '18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 57–65. ISBN: 9781450356299. DOI: 10.1145/3185768.3185775. URL: https://doi.org/10.1145/3185768.3185775.

[24] Vladimir Galabov. *A historic data center quarter with over 15% of servers running on AMD*. 2021-09-10. URL: https://omdia.tech.informa.com/blogs/2021/a-historic-data-center-quarter-with-over-15-of-servers-running-on-amd (visited on 2021-10-04).

[25] Michael Kan. *AMD's Share of x86 CPU Market Hits 14-Year High*. 2021-09-12. URL: https://uk.pcmag.com/processors/135049/amds-share-of-x86-cpu-market-hits-14-year-high (visited on 2021-10-04).

[26] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. Volume 3(3A, 3B, 3C, 3D & 3D): System Programming Guide. 2016.

[27] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (2018-03). ISSN: 2376-3639. DOI: 10.1145/3177754. URL: https://doi.org/10.1145/3177754.

[28] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. "Measuring Energy Consumption for Short Code Paths Using RAPL". In: *SIGMETRICS Perform. Eval. Rev.* 40.3 (2012-01), pp. 13–17. ISSN: 0163-5999. DOI: 10.1145/2425248.2425252. URL: https://doi.org/10.1145/2425248.2425252.

[29] Huazhe Zhang and H Hoffman. "A quantitative evaluation of the RAPL power control system". In: *Feedback Computing* 6 (2015).

[30] AM Devices. "Bios and kernel developer's guide (BKDG) for AMD family 15h models 00h–0Fh processors (2012)". In: *URL https://www.amd.com/system/files/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf* ().

[31] Robert Schöne, Thomas Ilsche, Mario Bielert, Markus Velten, Markus Schmidl, and Daniel Hackenberg. "Energy Efficiency Aspects of the AMD Zen 2 Architecture". In: *CoRR* abs/2108.00808 (2021). arXiv: 2108.00808. URL: https://arxiv.org/abs/2108.00808.

[32] EA Jagroep, Jan Martijn EM van der Werf, Jordy Broekman, Leen Blom, Rob van Vliet, and Sjaak Brinkkemper. "A resource utilization score for software energy consumption". In: *ICT for Sustainability 2016* (2016), pp. 19–28.

[33] Eva Kern, Markus Dick, Stefan Naumann, and Andreas Filler. "Labelling Sustainable Software Products and Websites: Ideas, Approaches, and Challenges." In: *EnviroInfo/ICT4S (1)*. 2015, pp. 82–91.

[34] Ziliang Zong, Rong Ge, and Qijun Gu. "Marcher: A heterogeneous system supporting energy-aware high performance computing and big data analytics". In: *Big data research* 8 (2017), pp. 27–38.

[35] Gustavo Pinto, Fernando Castor, and Yu David Liu. "Mining questions about software energy consumption". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. 2014, pp. 22–31.

[36] Georgios Kalaitzoglou, Magiel Bruntink, and Joost Visser. "A practical model for evaluating the energy efficiency of software applications". In: (2014).

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

[38] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. "Ranking programming languages by energy efficiency". In: *Science of Computer Programming* 205 (2021), p. 102609. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102609. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[39] Stefanos Georgiou, Maria Kechagia, Panos Louridas, and Diomidis Spinellis. "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.

[40] Gilson Rocha, Fernando Castor, and Gustavo Pinto. "Comprehending Energy Behaviors of Java I/O APIs". In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2019, pp. 1–12. DOI: 10.1109/ESEM.2019.8870158.

[41] Wellington Oliveira, Renato Oliveira, Fernando Castor, Benito Fernandes, and Gustavo Pinto. "Recommending Energy-Efficient Java Collections". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 160–170. DOI: 10.1109/MSR.2019.00033.

[42] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. "Post-Compiler Software Optimization for Reducing Energy". In: *SIGARCH Comput. Archit. News* 42.1 (2014-02), pp. 639–652. ISSN: 0163-5964. DOI: 10.1145/2654822.2541980. URL: https://doi-org.zorac.aub.aau.dk/10.1145/2654822.2541980.

[43] Luis Cruz, Rui Abreu, John Grundy, Li Li, and Xin Xia. "Do Energy-Oriented Changes Hinder Maintainability?" In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 29–40. DOI: 10.1109/ICSME.2019.00013.

[44] Cagri Sahin, Lori Pollock, and James Clause. "How Do Code Refactorings Affect Energy Usage?" In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '14. Torino, Italy: Association for Computing Machinery, 2014. ISBN: 9781450327749. DOI: 10.1145/2652524.2652538. URL: https://doi-org.zorac.aub.aau.dk/10.1145/2652524.2652538.

[45] Marco Couto, João Saraiva, and João Paulo 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.

[46] Déaglán Connolly Bree and Mel Ó Cinnéide. "Inheritance versus Delegation: Which is More Energy Efficient?" In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. ICSEW'20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 323–329. ISBN: 9781450379632. DOI: 10.1145/3387940.3392192. URL: https://doi.org/10.1145/3387940.3392192.

[47] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[48] Mike Mol. *Rosetta Code*. 2021. URL: http://www.rosettacode.org/wiki/Rosetta_Code (visited on 2021-09-23).

[49] Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. "jstanley: Placing a green thumb on java collections". In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2018, pp. 856–859.

[50] Irene Manotas, Lori Pollock, and James Clause. "SEEDS: A Software Engineer's Energy-Optimization Decision Support Framework". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 503–514. ISBN: 9781450327565. DOI: 10.1145/2568225.2568297. URL: https://doi.org/10.1145/2568225.2568297.

[51] Jacob Ruberg Nørhave, Casper Susgaard Nielsen, and Anne Benedicte Abildgaard Ejsing. *IDE Extension for Reasoning About Energy Consumption*. MA Thesis, Aalborg University, 2021.

[52] Oracle. *Variables*. URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html (visited on 2021-09-29).

[53] Oracle. *String*. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/String.html (visited on 2021-09-29).

[54] Oracle. *The Catch or Specify Requirement*. URL: https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html (visited on 2021-09-29).

[55] Oracle. *Arrays*. URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html (visited on 2021-09-29).

[56] Oracle. *Equality, Relational, and Conditional Operators*. URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html (visited on 2021-09-29).

[57] Oracle. *The for Statement*. URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html (visited on 2021-09-29).

[58] Oracle. *Operators*. URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html (visited on 2021-09-29).

[59] Oracle. *Control Flow Statements*. URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html (visited on 2021-09-29).

[60] Oracle. *Package java.lang*. URL: https://docs.oracle.com/javase/9/docs/api/java/lang/package-summary.html (visited on 2021-09-29).

[61] Oracle. *Autoboxing and Unboxing*. URL: https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html (visited on 2021-09-29).

[62] Oracle. *Class Thread*. URL: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Thread.html (visited on 2021-09-29).

[63] Microsoft. *Language Integrated Query (LINQ) (C#)*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/ (visited on 2021-10-13).

[64] Microsoft. *Boxing and Unboxing (C# Programming Guide)*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing (visited on 2021-10-06).

[65] Microsoft. *try-catch*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch (visited on 2021-10-07).

[66] Microsoft. *try-finally*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-finally (visited on 2021-10-07).

[67] Microsoft. *try-catch-finally*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch-finally (visited on 2021-10-07).

[68] Microsoft. *Delegates (C# Programming Guide)*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/ (visited on 2021-10-04).

[69] Microsoft. *Action Delegate*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/api/system.action?view=net-5.0 (visited on 2021-10-04).

[70] Microsoft. *Func<TResult> Delegate*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/api/system.func-1?view=net-5.0 (visited on 2021-10-04).

[71] Microsoft. *MethodBase.Invoke Method*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.methodbase.invoke?view=net-5.0 (visited on 2021-10-18).

[72] Microsoft. *Init Only Setters*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/init (visited on 2021-10-06).

[73] Microsoft. *Constants (C# Programming Guide)*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constants (visited on 2021-10-06).

[74] Microsoft. *readonly (C# Reference)*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/readonly (visited on 2021-10-06).

[75] Microsoft. *Properties (C# Programming Guide)*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties (visited on 2021-10-06).

[76] Microsoft. *Object Class*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/api/system.object?view=net-5.0 (visited on 2021-10-06).

[77] Microsoft. *Structs*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/structs (visited on 2021-10-06).

[78] Microsoft. *Records*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record (visited on 2021-10-06).

[79] Microsoft. *Inheritance - derive types to create more specialized behavior*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/inheritance (visited on 2021-12-15).

[80] Microsoft. *interface*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface (visited on 2021-10-06).

[81] Microsoft. *abstract*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract (visited on 2021-10-06).

[82] Microsoft. *default interface methods*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods (visited on 2021-10-06).

[83] Bill Wagner Microsoft. *Generic classes and methods*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics (visited on 2021-10-06).

[84] Microsoft. *break*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/break (visited on 2021-10-06).

[85] Microsoft. *continue*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/continue (visited on 2021-10-06).

[86] Microsoft. *return*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/return (visited on 2021-10-06).

[87] Microsoft. *goto*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/goto (visited on 2021-10-06).

[88] Jóakim von Kistowski, Jeremy Arnold, Karl Huppler, Klaus-Dieter Lange, John Henning, and Paul Cao. "How to Build a Benchmark". In: *ICPE 2015 - Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (2015-02). DOI: 10.1145/2668930.2688819.

[89] Noah Gibbs. *Microbenchmarks vs Macrobenchmarks (i.e. What's a Microbenchmark?)* 2019. URL: https://engineering.appfolio.com/appfolio-engineering/2019/1/7/microbenchmarks-vs-macrobenchmarks-ie-whats-a-microbenchmark (visited on 2021-10-06).

[90] Nicolas Poggi. "Microbenchmark". In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Y. Zomaya. Cham: Springer International Publishing, 2019, pp. 1143–1152. ISBN: 978-3-319-77525-8. DOI: 10.1007/978-3-319-77525-8_111. URL: https://doi.org/10.1007/978-3-319-77525-8_111.

[91] Sourceforge. *The Complete Open-Source and Business Software Platform*. URL: https://sourceforge.net (visited on 2021-10-11).

[92] CLBG. *Computer Language Benchmark Game*. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html (visited on 2021-10-11).

[93] Manish Bhojasia. *Sanfoundry C# Programming Examples*. URL: https://www.sanfoundry.com/csharp-programming-examples/ (visited on 2021-10-11).

[94] Include Help. *C# Basic Programs*. URL: https://www.includehelp.com/dot-net/basic-programs-in-c-sharp.aspx (visited on 2021-10-11).

[95] Standard Performance Evaluation Corporation. *Spec Benchmark Suite Prices*. URL: https://www.spec.org/order.html (visited on 2021-10-14).

[96] Peter Sestoft. "Microbenchmarks in Java and C#". In: *Lecture Notes, September* (2013).

[97] Stan Brown. "How big a sample do I need?" In: *Brownmath* (2013).

[98] Alexander Brandborg, Michael Wit Dolatko, Anders Munkgaard, and Claus Worm Wiingreen. *PapaGo*. Aalborg University, 2018. URL: https://projekter.aau.dk/projekter/files/281491858/final.pdf.

[99] Dr. Saul McLeod. *What a p-value tells you about statistical significance*. 2019. URL: https://www.simplypsychology.org/p-value.html (visited on 2021-09-23).

[100] Microsoft. *ReadyToRun Compilation*. 2021-09-15. URL: https://docs.microsoft.com/en-us/dotnet/core/deploying/ready-to-run (visited on 2021-10-15).

[101] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical software engineering* 14.2 (2009), pp. 131–164.

[102] Roger-luo. *Reproducible benchmarking in Linux-based environments*. 2021-05-19. URL: https://github.com/JuliaCI/BenchmarkTools.jl/blob/863c514f559cab04a05315e84868183fbfa8758d/docs/src/linuxtips.md (visited on 2021-09-23).

[103] Aleksander Øster Nielsen, Kasper Jepsen, Lasse Stig Emil Rasmussen, Milton Kristian Lindof, Rasmus Smit Lindholt, Søren Bech Christensen, Lars Rechter, Martin Jensen, Casper Susgaard Nielsen, Anne Benedicte Abildgaard Ejsing, and Jacob Ruberg Nørhave. *CsharpRAPL*. URL: https://gitlab.com/Plagiatdrengene/CsharpRAPL/.

[104] Ryan Smith. *Intel Launches Comet Lake-U and Comet Lake-Y: Up To 6 Cores for Thin & Light Laptops*. 2019-08-21. URL: https://www.anandtech.com/show/14782/intel-launches-comet-lakeu-and-comet-lakey-10th-gen-core-for-low-power-laptops (visited on 2021-11-01).

[105] Ian Cutress. *Intel's 10th Gen Comet Lake for Desktops: Skylake-S Hits 10 Cores and 5.3 GHz*. 2019-04-30. URL: https://www.anandtech.com/show/15758/intels-10th-gen-comet-lake-desktop (visited on 2021-11-01).

[106] BenchmarkDotNet. *Good Practices*. URL: https://benchmarkdotnet.org/articles/guides/good-practices.html (visited on 2021-11-17).

[107] accord-net. *Accord.NET Machine Learning Framework*. URL: http://accord-framework.net/ (visited on 2021-10-18).

[108] Intel. *Intel® Xeon® W-1250P Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/199340/intel-xeon-w1250p-processor-12m-cache-4-10-ghz.html (visited on 2021-12-27).

[109] Microsoft. *BindingFlags Enum*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.bindingflags?view=net-6.0 (visited on 2021-12-16).

[110] Microsoft. *C# documentation*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/ (visited on 2021-12-09).

[111] ECMA International. *Common Language Infrastructure (CLI) Partitions I to VI*. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-335_6th_edition_june_2012.pdf (visited on 2021-11-23).

[112] Jetbrains. *Rider*. URL: https://www.jetbrains.com/rider/ (visited on 2021-12-09).

[113] Jetbrains. *View Intermediate Language (IL)*. URL: https://www.jetbrains.com/help/rider/Viewing_Intermediate_Language.html (visited on 2021-12-09).

[114] Microsoft. *Visual Studio*. URL: https://visualstudio.microsoft.com/ (visited on 2021-12-09).

[115] Microsoft. *View disassembly code*. URL: https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-use-the-disassembly-window?view=vs-2022 (visited on 2021-12-09).

[116] Microsoft. *Integral numeric types (C# reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types (visited on 2021-11-23).

[117] Microsoft. *?: operator (C# reference)*. URL: https://github.com/icsharpcode/ILSpy (visited on 2021-12-15).

[118] Microsoft. *Selection statements (C# reference)*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/selection-statements (visited on 2021-09-16).

[119] Microsoft. *ImmutableList<T> Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablelist-1?view=net-6.0 (visited on 2021-12-15).

[120] Microsoft. *LinkedList<T>.Remove Method*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1.remove?view=net-6.0 (visited on 2021-12-14).

[121] Microsoft. *.NET Performance Tips*. URL: https://docs.microsoft.com/en-us/dotnet/framework/performance/performance-tips#boxing-and-unboxing (visited on 2021-12-14).

[122] Microsoft. *ReadOnlyDictionary<TKey,TValue> Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.readonlydictionary-2?view=net-6.0 (visited on 2021-12-15).

[123] Microsoft. *Dictionary<TKey,TValue> Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-6.0 (visited on 2021-12-15).

[124] Microsoft. *String.Format Q & A*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.string.format?view=net-6.0%5C#stringformat-q--a (visited on 2021-12-07).

[125] Microsoft. *$ - string interpolation (C# reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated (visited on 2021-12-07).

[126] Microsoft. *StringBuilder Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.text.stringbuilder?view=net-6.0#HowWorks (visited on 2021-12-07).

[127] Microsoft. *Local functions (C# Programming Guide)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/local-functions (visited on 2021-12-15).

[128] Microsoft. *virtual (C# Reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual (visited on 2021-12-13).

[129] Microsoft. *Exceptions*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/exceptions (visited on 2021-12-09).

[130] Microsoft. *try-catch (C# Reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch (visited on 2021-12-09).

[131] Microsoft. *ArgumentException Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.argumentexception?view=net-6.0 (visited on 2021-12-09).

[132] Microsoft. *DivideByZeroException Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.dividebyzeroexception?view=net-6.0 (visited on 2021-12-09).

[133] ScottPlot. *ScottPlot*. URL: https://scottplot.net/ (visited on 2021-10-25).

[134] Microsoft. *dotnet/roslyn*. URL: https://github.com/dotnet/roslyn (visited on 2021-12-15).

[135] Microsoft. *dotnet/runtime*. URL: https://github.com/dotnet/runtime (visited on 2021-12-15).

Appendix A

Appendix

A.1 Preliminary Testing

Important to note: Just-In-Time (JIT) compilation is the default compilation for C#.

In the preliminary testing, each benchmark has 10.000 iterations and are run 500 times. When the column says first removed, it means the first result of the 500 is removed. Each result is the average of the 500 runs.

A.1.1 Preliminary Testing using JIT compilation

Benchmark	Time (ms)	Time - First Removed (ms)
Add	0.0493	0.0491
Unrolled Add	0.0904	0.0480
Subtract	0.0540	0.0532
Unrolled Subtract	0.0959	0.0540
Division	0.0482	0.0480
Unrolled Division	0.0897	0.0467
Multiplication	0.0478	0.0477
Unrolled Multiplication	0.0912	0.0489

These results are used in Section 4.5, as basis for why loops are used instead of unrolling.

A.1.2 Preliminary Testing using ReadyToRun (R2R) compilation

Benchmark	Time (ms)	Time - First Removed (ms)
Add	0.0380	0.0378
Unrolled Add	0.0866	0.0480
Subtract	0.0425	0.0419
Unrolled Subtract	0.0921	0.0522
Division	0.0381	0.0379
Unrolled Division	0.0901	0.0505
Multiplication	0.0376	0.0374
Unrolled Multiplication	0.0915	0.0516

These results are used in Section 4.5, as basis for why loops are used instead of unrolling.

A.1.3 Preliminary Testing Comparing For and While Loops with JIT Compilation

Benchmark	For Loop Time (ms)	For Loop Time (ms) First Removed	While Loop Time (ms)	While Loop Time (ms) First Removed
Add	0.0493	0.0491	0.0469	0.0468
Subtract	0.0540	0.0532	0.0490	0.0488
Division	0.0482	0.0480	0.0484	0.0483
Multiplication	0.0478	0.0477	0.0474	0.0472

These results are used in Section 4.5, as basis for why a `for` loop is used as there are no significant differences between `for` and `while` loops.

A.1.4 Preliminary Testing using an empty loop

Benchmark	Time (ms)
Empty Loop (1.000.000 iterations)	0,244667
Empty Benchmark	<0,01

These results show that the compiler does not optimize an empty loop away, so in cases where the compiler optimizes something away, it is just the instructions inside the loop.

A.1.5 Preliminary Testing Overflow

Benchmark	Time (ms)
Overflow (1.000.000 iterations)	0,310782
No Overflow (1.000.000 iterations)	0,31132

These results show that overflow does not cost anything. Furthermore, the p-values between these numbers are above 0,05 meaning the difference between the two are not significant. Lastly, the p-values between these and the empty loop is pretty much as close to 0 as possible (Our calculation says 2,22E-16), meaning they are significantly different from an empty loop and therefore the compiler does not optimize the code away. This is important when designing benchmarks, as this shows we do not need to worry about variables overflowing in the benchmarks.

A.1.6 Preliminary Testing using an empty loop with and without warmup

Benchmark	Time (ms)
Empty Loop With Warmup (1.000.000 iterations)	0,244718
Empty Loop W/O Warmup (1.000.000 iterations)	0,244722

These results show that warmup does not have a big impact on the average result of a benchmark, in fact, the p-value is 0,966273 meaning there is a high chance that they are the same.

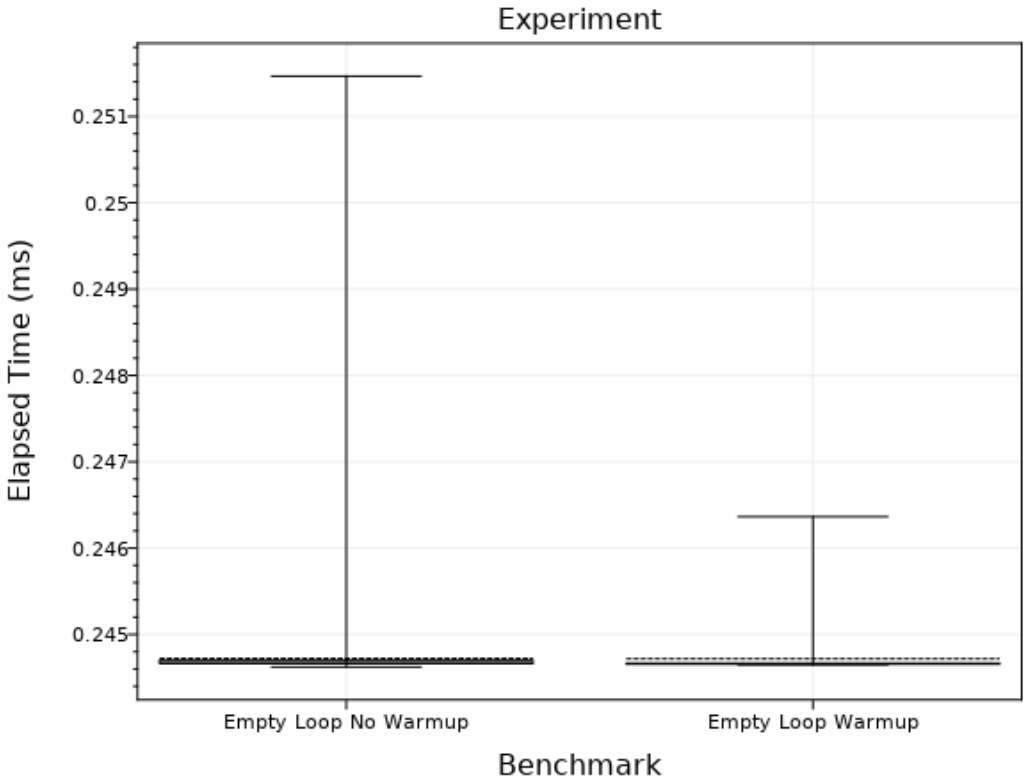

Figure A.1: Boxplot showing the elapsed time for this experiment.

The results seen in Figure A.1 show that there are significant outliers when not warming up, which is unnecessary, therefore we use warmup, despite the average result converging to the same number. This is used in Section A.2.

A.1.7 Preliminary Testing showing that the results approximates a normal distribution

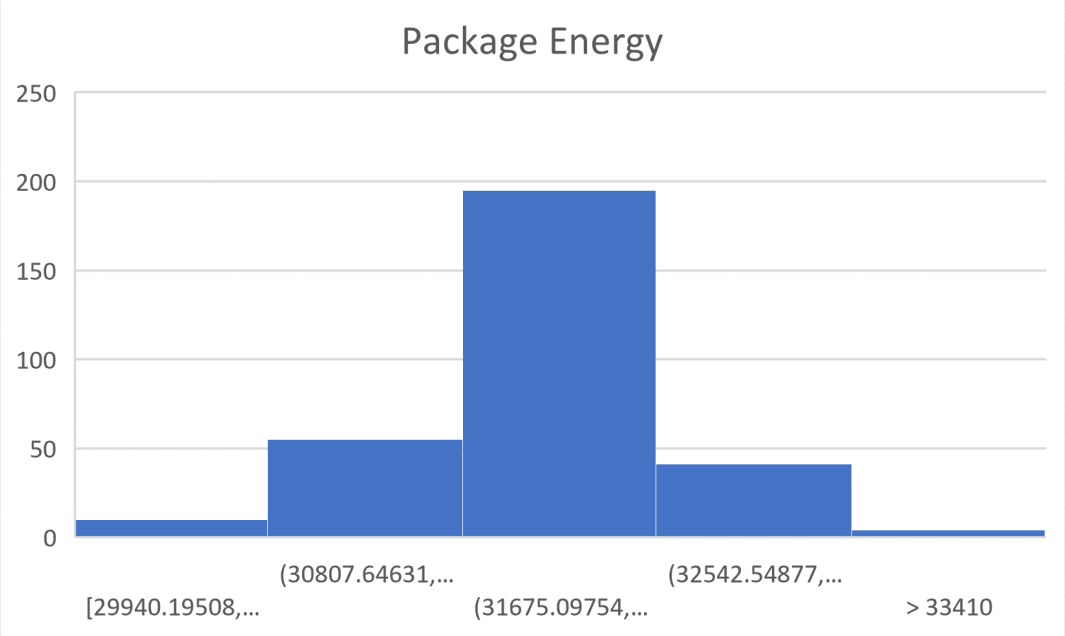

Figure A.2: Bargraph showing that results approximate a normal distribution.

The results seen in Figure A.2 show that results of a benchmark approximates a normal distribution. This is used in Section 4.5.

A.1.8 Preliminary Testing of previously created benchmarks

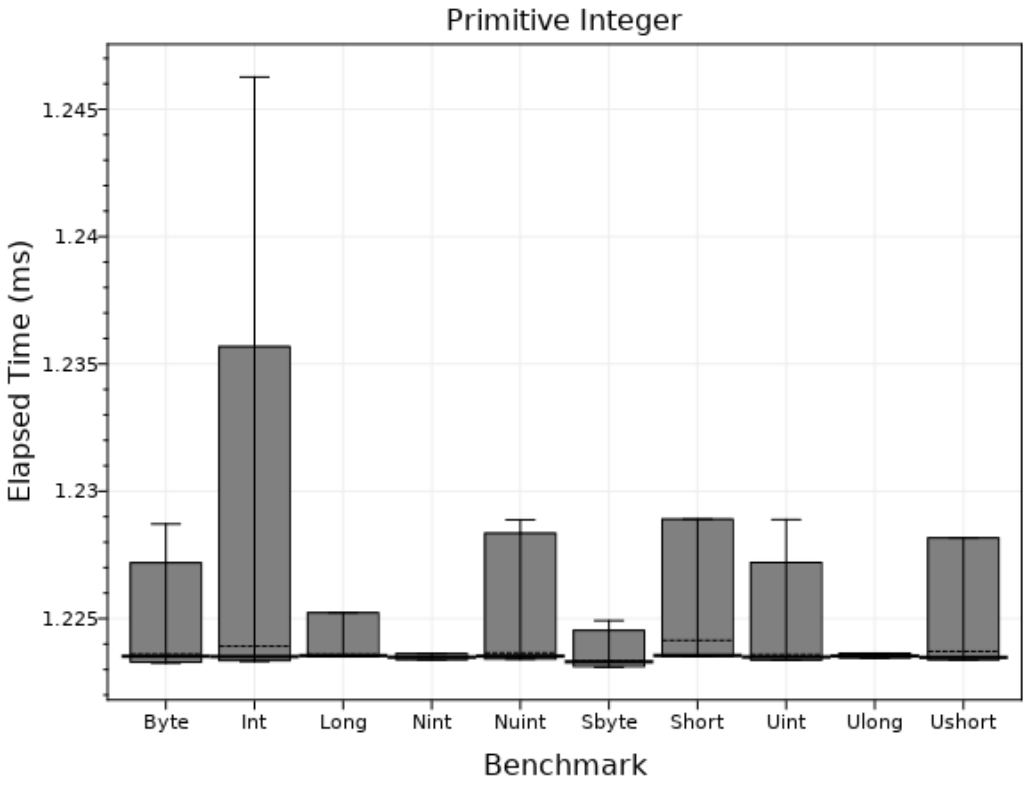

Figure A.3: Boxplot showing the elapsed time for the primitive integer types in [11].

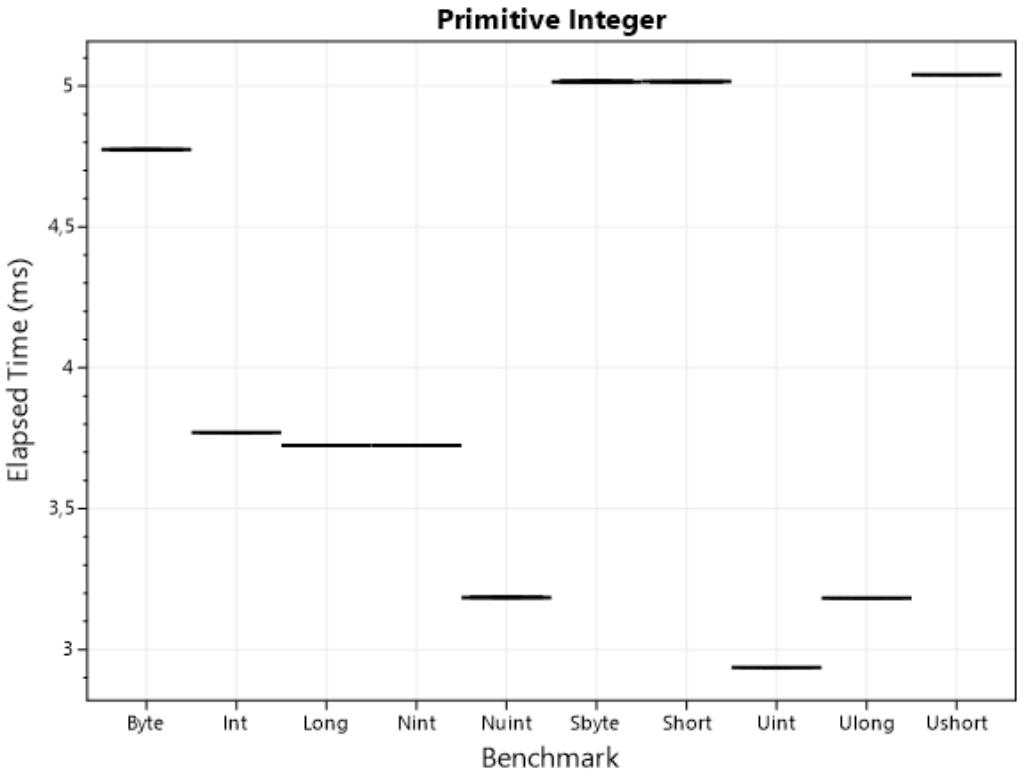

Figure A.4: Boxplot showing the elapsed time for the primitive integer types in our benchmarks.

The results seen in Figure A.3 show that when implementing the benchmarks shown in [11], in C# there are large issues with outliers and compiler optimization. From the plot itself, it seems like every type of primitive integer is approximately equally efficient with regards to elapsed time, while other experiments that also test primitive integer types have shown other results, for example seen in Figure A.4.

```
1 long a = 0;
2
3 for (long j = 1; j < iter; j++) {
4     for (long k = 1; k < iter; k++) {
5         a = j + k;
6     }
7 }
```

Listing 31: Code testing add operation in [11].

The code seen in Listing 31 can not be compared with similar code that would utilize the `const` keyword, as all three variables needs to be changed for this benchmark to work. This is another issue that exists with the benchmarks shown in [11] and [4], and another reason we create our own type of benchmarks.

These issues are found for all the benchmarks we have tested from [11] and [4], therefore we do not use these benchmarks and instead create our own. This is used in Table 5.1

A.2 Full Implementation Explanation

The full implementation explanation has been moved to our `GitLab` [103] in the Appendix PDF.

A.3 All Results

The results for all of the benchmarks has been moved to our `GitLab` [103] in the Appendix PDF.

A.4 Full Analysis

The full analysis has been moved to our `GitLab` [103] in the Appendix PDF.

A.5 Benchmark Variance

Concurrent Dictionary Insertion	Approximate Median Package Energy (μJ)
Run 1	776.000.000
Run 2	777.000.000
Run 3	771.000.000
Run 4	776.000.000
Run 5	775.000.000
Run 6	769.000.000
Run 7	775.000.000
Run 8	775.000.000
Run 9	777.000.000
Run 10	774.000.000

These results show that the same benchmark can have a variance of slightly above 1% in median Package Energy between runs. We have observed this for many benchmarks and have observed that just because one benchmark has results in the lower end of its variance, another benchmark may have a result in the higher end of its variance. The variance between the iterations in each benchmark during one run is small enough that the same benchmark can have a significant difference (p-value below 0,05) from itself between runs. The only benchmark where we have observed higher variance is for the inheritance benchmarks. This is used in Section 8.2.

A.6 Inheritance Variance

Package Energy (μJ)	Abstract	Inheritance	Inheritance Virtual	Inheritance Virtual Override	Interface	No Inheritance
Run 1	18.800	21.000	17.400	17.400	17.400	21.200
Run 2	19.600	21.400	20.200	20.200	19.600	21.600
Run 3	17.400	20.800	19.400	17.400	19.200	20.800
Run 4	17.300	20.600	19.600	17.400	18.800	21.200
Run 5	18.500	20.700	19.200	17.300	18.500	21.000
Run 6	17.400	21.000	17.400	17.500	17.400	21.200
Run 7	17.400	22.600	17.500	19.600	19.800	22.600
Run 8	19.500	20.800	17.400	17.300	17.400	22.200
Run 9	17.400	22.600	17.400	19.400	19.300	22.600
Run 10	17.300	21.000	19.500	17.100	19.400	20.600

These results show that the same benchmark within inheritance can have very different results between runs. The variance within the iterations in

each benchmark during one run is small enough that the same benchmark can have a significant difference (p-value below 0,05) from itself between runs. From these results we can see that the results can vary by up to 16,76% in median Package Energy in the worst scenarios for inheritance between runs. This is used in Chapter 10.