Improving C# Programs for Energy Efficiency

Helping Developers write Energy Efficient Code

Specialization in Software

Project Report
CS-22-PT-10-06

Aalborg University
Software

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

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Improving C# Programs for Energy Efficiency

Theme:
Energy Aware Programming

Project Period:
Spring Semester 2022

Project Group:
CS-22-PT-10-06

Participant(s):
Lasse Stig Emil Rasmussen
Milton Kristian Lindof
Søren Bech Christensen

Supervisor(s):
Bent Thomsen, Lone Leth Thomsen

Copies: 1

Page Numbers: 157

Date of Completion:
June 8, 2022

Abstract:

This project builds upon our research into the energy consumption of language constructs in C# to create a linter called *Energy Analyzer*. First, we create microbenchmarks, we use the results from these microbenchmarks and the microbenchmarks from our previous research, and apply them to larger benchmarks to see if the results can be generalized into larger programs. We use the results from the larger benchmarks to create *Energy Analyzer*, which is a NuGet package that aims to help developers write more energy efficient C# code. `Energy Analyzer` consists of two parts, a suggestion part, which finds C# constructs with energy efficient alternatives and gives the suggestion to change the construct to the energy efficient alternative. The second part is the code fix part, which, if chosen by the developer, automatically changes the construct to a more energy efficient one. We evaluate `Energy Analyzer` in both performance and usability, where we find 15,05% energy consumption can be saved in one benchmark, while no significant difference can be found in another benchmark, while the usability was satisfying for the tested developers.

Preface

This project has been created by 3 software students in the 10th semester, at Aalborg University, under the theme *Energy Aware Programming* in the period 1st of February 2022 to the 10th of June 2022. We want to thank our supervisors Bent Thomsen and Lone Leth Thomsen for their guidance throughout this project.
The code created during this project is freely available at [1] and [2], with the CSharpRAPL framework created in [3] being freely available at [4].
The NuGet package with the linter created in this project, is available at [5], and the NuGet package with the CSharpRAPL framework created in [3], is available at [6].

Summary

This project involves researching language constructs in C# and how they affect the energy usage of programs. We use this knowledge to create a linter called *Energy Analyzer*. *Energy Analyzer* is a Roslyn analyzer that utilizes the research done in this project, as well as our previous project [3], to give suggestions and code fixes to developers that may help decrease the energy usage of their software.

Researching and understanding the results of the language constructs is sometimes a challenge as there is uncertainty when measuring the energy consumption of software. To figure out why there is a lot of variance in some of our results we go through steps including reading the documentation behind the language constructs, reading the library implementations, etc. Getting an understanding of the results help us create more accurate suggestions for developers to save energy, and therefore better suggestions for *Energy Analyzer*.

Besides this, we generalize the results of the microbenchmarks to larger benchmarks making it possible for us to figure out in what circumstances the language constructs may be the most efficient, and when they might not be as efficient as our microbenchmarks showed. Here we found interesting results, among these, that `string` interpolation is the most efficient type of concatenation when only two `strings` are concatenated, that `for` loops are more efficient than `foreach` loops when the index of an `array` is needed etc.

Following the results from generalizing microbenchmarks to larger benchmarks, we create formal suggestions that is implemented in *Energy Analyzer*. The formal suggestions are the ones shown to developers when using *Energy Analyzer* and is used to save energy consumption when programming.

Creating *Energy Analyzer* is a learning experience, as we have no previous experience creating Roslyn analyzers. We overcome this obstacle by implementing a small part of *Energy Analyzer*, ensuring it works and then going to the next part of it, making us able to understand how they are built so we can create *Energy Analyzer* properly.

Despite following this path, there is a lot of challenges with regards to bugs where *Energy Analyzer* finds wrong places to create suggestions, not creating suggestions at all, or creating incorrect code fixes.

Most of these issues are resolved and we have a linter that is ready to be utilized by developers, however, according to our evaluation, there may be cases where the developer should be careful blindly following *Energy Ana-*

lyzer as the efficiency improvements are not large enough to justify making the code less readable.

The usability tests done with *Energy Analyzer* suggests that the linter is easy to use, therefore no changes need to be made to make it possible for other people to utilize *Energy Analyzer*.

There are improvements that can be made to *Energy Analyzer* in the future, such as creating more suggestions, creating code fixes for the rest of the suggestions, and generalizing more of the language constructs to larger benchmarks, and to a higher variety of benchmarks, so that more use cases are covered in the research.

Besides this, *Energy Analyzer* could be extended to work as an Integrated Development Environment (IDE) extension instead of mainly a NuGet package, as that would create further opportunities to decrease energy consumption.

Contents

Chapter 1

Introduction

Energy Consumption from Information and Communications Technology (ICT) systems have increased in recent years. In [7] it is reported that the energy consumption of data centers has grown by about 6% from 2010 to 2018, despite significant energy efficiency increases in processing units. The amount of energy the data centers consume is expected to increase in the coming years, with one of the most worrying forecasts showing an increase of ICT energy consumption from around 2% of global energy consumption to somewhere around 20% within the next 15 years [8]. This includes data centers which alone may account for 8% of global energy consumption by 2030 [8]. This means further improvements in energy efficiency within ICT are needed.

Such improvements in energy efficiency within ICT can be achieved in different ways, one of these ways is writing energy efficient software. Research within this has shown that there are a lot of different ways of improving software, such as choosing the right programming language, choosing the right language constructs, using the right algorithms, using the right coding style etc. [9, 10, 11, 12, 3, 13]

When looking within the research of energy efficiency in software, there is a distinct lack of tools that can help developers write energy-efficient code, with the most notable tools being for Java or being for measuring energy consumption instead of helping developers choose the right constructs [3, 14, 15, 16, 17].

Previous work is focused mainly on Java, and as we seek to expand the field to help developers outside of the Java community, we focus on another language. According to [18], C# is the second most used Object-Oriented Programming (OOP) language and is on the rise in popularity.

This, combined with our previous work [3] being for C#, means we choose to focus on C#.

Furthermore, despite the research and our previous work [3] within energy consumption in software, and especially C#, the information is sparse when considering how the code should be written. In addition, most previous research focuses on microbenchmarks and does not look further into if these findings generalized to larger benchmarks.

Given that there still are a lot of language constructs that have not been measured with regards to energy efficiency, for example within the group of invocation there is a lack of research with a focus on lambda expressions done in [3], this is an area for further research. The same group of language constructs is defined so that the outcome of the code that uses a language construct from a group should be equivalent if possible, however, this is not always possible, in which case the output should be similar [3]. Furthermore, the area of `Exceptions` can be expanded upon, as we can split the benchmarks in [3] into multiple benchmarks to get better insight into which part of creating, throwing, and catching `Exceptions` has the most significant energy cost.

Besides the area of individual language constructs, there is a lack of research into generalizing energy efficiency of language constructs to larger programs, and a lack of tools for C# that can help developers choose the right constructs.

Getting more knowledge of individual language constructs and how these can be generalized to a larger program can be used to create a tool that can help developers write energy-efficient code.

When using the knowledge of specific language constructs, it is natural to analyze the patterns of code that are written to find sub-optimal language construct usage. When creating a tool that analyzes patterns of code, a linter is a natural choice as it lends itself well to recognizing patterns from the program's abstract syntax tree. Linters are also design-time tools, meaning they are meant to help guide the developer when writing the code. A linter can utilize the knowledge we gather to improve code [19], therefore this is the focus of our research.

1.1 Problem Statement

This section presents the problem statement and research questions for the project and the motivation behind it.

How can developers write more energy efficient code using C# with knowledge of the energy consumption of language constructs?

1. *How can we extend upon the microbenchmarks from previous language constructs energy efficiency research?*
2. *How can the results from the microbenchmarks be generalized to larger benchmarks?*
3. *How can these results be used to create a linter to help developers improve the energy efficiency of their programs?*

The problem statement and research questions are motivated by a lack of research with regards to language constructs in C#, and how microbenchmarks created and measured in previous work affect larger programs. Furthermore, the problem statement and research questions are motivated by a lack of tools, that can help developers write energy-efficient code in C#. A need for tools that help developers write energy efficient code is needed according to [20, 21], therefore the main contribution of this project is a linter that can help developers write energy efficient code by hinting towards more energy efficient language constructs.

As the main reasons behind the need for increased energy awareness are the popularity of mobile devices, cloud computing, embedded systems, and data center-based services [20, 21], we choose to focus on making it easy for all the developers working on specific applications for these systems to create energy efficient code. We choose to focus on applications instead of the developers, as we believe that focusing on applications can have a greater impact compared to individual developers having to install a linter. We believe this is the case because large projects could be looked through by multiple developers, who would all have to install the linter. However, if the linter is project-specific, only the project would need the linter and the developers do not need to install anything to their Integrated Development Environment (IDE).

The research conducted in [3] shows that different language constructs and collections have different amounts of energy consumption. As we want to create a linter, we need to extend our knowledge base of language constructs, so we can have accurate suggestions for developers. Furthermore, we need to ensure that the results can be generalized, because otherwise, the linter may not be able to give accurate suggestions.

Following the research questions, we split the work into three subjects:

1. Extend the previous work of language constructs with lambda expressions and more exception analysis,

2. Get an understanding of how research regarding language constructs can be generalized into larger programs, and

3. Create a linter that can help developers write energy-efficient code.

The linter will serve to help developers write more efficient code during the writing process. Furthermore, the linter will be able to analyze an existing set of code and give insight into where there could be improvements to be made, based on the research done in [3] and in this project. The linter will be evaluated by how easy it is to use and how much the changes made with the linter affect the energy consumption of different software.

When creating a linter for C#, a Roslyn Analyzer is a natural choice, as the .NET compiler platform uses this to inspect the C# code [22], and Microsoft has a guide on how to make a Roslyn analyzer [23]. The linter will be giving suggestions and code fixes for developers' code with regards to energy consumption, based on the research done previously regarding energy consumption, therefore we call the linter *Energy Analyzer*.

The specifics of how to implement a Roslyn Analyzer is explained in Section 5.2.

We examine related work in Section 1.3, to get an overview of what already exists within this field. After this, we describe the design choices of our benchmarks in Chapter 2, which brings us to the results of our benchmarks in Chapter 3, which serves to get a starting point with regards to suggestions for *Energy Analyzer*. After presenting the results of our benchmarks, we analyze them to figure out why they occur in the way they do in Chapter 4, this also serves to create formal suggestions for *Energy Analyzer*. *Energy Analyzer* is designed, implemented, and evaluated in Chapter 5, using the formal suggestions gathered from the benchmark analysis. Once *Energy Analyzer* is evaluated, we reflect over the project in Chapter 6, to get an understanding of what went right and what could have been better throughout the project. The reflections lead us to the conclusion of the project in Chapter 7, after which we end off by presenting what future work could be done within this field in Chapter 8.

1.2 Work Process

In this section, we discuss our work process. We use a hybrid development process between a plan-driven and an agile process [24]. We describe it as a hybrid because we first create a plan for the three subjects we want to research, for each subject we iterate through it until we have satisfactory results, and we then proceed to the next subject of our study.

This hybrid process allows us to adapt to unforeseen challenges or inspiration that may show up throughout the project. This project spans approximately a four-month duration. Because of this, we can not make many iterations over our product.

We do not use a purely plan-driven process, as we can not plan the entire project from the start, because a lot of variables can change during the project, such as which subjects are needed to study during the project, or how in-depth different subjects needs to be studied, therefore changing the direction of the project. Therefore, a similar process to the agile process is used, where we plan the near future and have ideas for what comes later in the project. We have deadlines to make sure that we meet the hard deadline for the project.

The report is iterated over multiple times as we review our work and when we get feedback from our supervisors. We have daily stand-up meetings to keep an overview of the status of the project, share information, and to keep track of what each other are working on. The overview is done to maintain an outline of what must be done in the project for the next days and to keep track of the timeline for the project. We discuss if anyone is having problems that the other two members can help solve.

We use Notion as a tool to keep track of what each other is doing [25]. The board contains four columns with cards with the tasks required for the project, to which one or more group members can be assigned to. The cards can be moved between the four columns depending on the state of the task. The four columns are To Do, Doing, Review, and Done. We use `Gitlab` [26] to keep track of our code repository, to be able to share and keep track of changes of the code written doing the project.

1.3 Related Work

In this section, we discuss related works to this project. We start by describing works that benchmark language constructs. Then we examine existing

tools that can help developers write more energy aware code.

An overview of related work has been created in our previous work [3], where we show research that has been done within the field of software energy efficiency. Because of this, we only highlight some papers from previous work and add a few papers that were not relevant in previous work.

1.3.1 Language Constructs

In [3], we present a framework that can be used for measuring C# code with regards to elapsed time and energy consumption, together with 316 microbenchmarks that gives an overview of a part of the language constructs that can be used in C#. Furthermore, we present the results of measuring these microbenchmarks, making it possible to use these results for further research, for example in generalizing these results for larger programs or for tools that can be used to help developers write more energy efficient code. We show that there are cases where there is a significant difference in energy consumption between language constructs that are similar, for example that `array` is the most efficient type of list datatype, with `List` being the second most efficient, and `Dictionary` being the most efficient type of table datatype among other results.

Another paper that looks at microbenchmarks is [10]. In this paper, a number of microbenchmarks and results from measuring these microbenchmarks in Java are presented. These results show that there are differences in the energy efficiency of language constructs that can be used for similar purposes, meaning that there is potential in this area.

The paper [27] extends the work of [10]. The paper presents more microbenchmarks and results from Java, further showing that there are potential improvements to be found within this area.

1.3.2 Tools

Several tools have been created to assist developers reason or optimize their programs for energy consumption. In [14] a tool capable of recommending Java collections to improve the energy consumption of the program is presented. They achieved this by creating an Eclipse plugin. The authors conclude that the plugin can optimize energy consumption by 2%-17% and the execution time by 2%-13%.

In [16] a framework called SEEDS, which aims to optimize the use of Java collections is presented. The framework takes a Java program as input and creates a new version of that program with suggested changes automatically. The authors conclude that their framework can optimize Java programs' energy consumption by 2%-17%.

Another tool is presented in [28]. A function-level profiling tool that measures the energy consumption, by performing program analysis during the program's run-time. The authors conclude that their tool can be used to profile the energy consumption of a program without modifying the program, therefore they conclude that it should be possible for the program to find energy hot spots. Besides this, they also show that the tool has a small amount of overhead.

The tool presented in [29] is an Eclipse plugin that estimates the energy consumption at program, function, and line level. The tool uses both program analysis and instruction energy modeling to achieve this estimation. The authors conclude that the tool can estimate the energy consumption to within 10% of the ground truth.

In [30] a tool for Android applications that is capable of detecting and refactoring energy-inefficient code is presented. They find that individual refactoring produces consistent gains but with varying amounts of effect. Furthermore, they find that combining refactorings, in general, reduces the energy consumption but not always, along with a few combinations that are harmful to reducing the energy consumption.

Another plugin is presented in [17] which can be used to measure programs written in Java in the Eclipse IDE. This plugin can also be used to help developers choose energy-efficient language constructs based on the research done in [10, 27]. The authors conclude that using their plugin can achieve up to 14,46% improvement in energy efficiency with up to 0,48% loss inaccuracy on the machine learning software "WEKA".

An IDE extension for Visual Studio Code is presented in [15], which is capable of giving the developers the estimated amount of energy their code would consume. The extension uses both static and dynamic analysis to estimate the energy consumption of the code. The authors conclude that non-linear machine learning estimates deviate less from the ground truth than both the energy model they look at and the linear machine learning models. The estimation approach with the least error was the random forest machine learning model which estimated between 7,49% below ground truth to 9,19% above ground truth, with a median of 1,06% above the ground truth, indicating a slight overestimation.

As we have looked at related work, we now look into how we should design our benchmarks to create further knowledge that is necessary for our linter.

Chapter 2

Benchmark Design

To get extra insight into language constructs within C#, and to improve upon the methodology presented in [3], we choose to create extra microbenchmarks for the categories *Exception* and *Invocation*. The reason behind improving the methodology is that there are small shortcomings of the methodology used in the previous project. These shortcomings are not enough to dismiss earlier results, however, they exist and could have an impact on future research within this area, therefore we create an improved formal methodology.

Specifically, we extend the *Exception* category by splitting *Exception* benchmarks into two: Creating an *Exception*, and throwing and catching an *Exception* and we extend the *Invocation* category by creating benchmarks for anonymous functions. We choose these categories to get insight into missed information regarding previously made microbenchmarks. The missed information is which part of *Exceptions* cost the most and if there are specifics when using *lambda expressions* in the *Invocation* category that should be taken note of.

We follow the same overall design and methodology as presented in [3], however, there are a few changes that improve the quality of the results and analysis. Specifically, we add a new step in the procedure [3, p.60] for presenting the results. The new step we add is doing a sanity check, meaning we check if the results are realistic. We also add a new step in the analysis process presented in [3, p.111] where we check if the construct we are testing is a library construct, in which case we check the implementation of the construct.

As mentioned, the design of microbenchmarks follows the design of our previous project [3], the design chapter can be found in Chapter 4 in [3],

therefore we only summarize each part of the design process. The design chapter is divided into seven parts, Goals, Microbenchmarks vs Macrobenchmarks, Choice of benchmarks, Measurement approach, Experiment Guidelines, Threats to Validity, and Summary.

2.1 Goals

The goal of the benchmarks is to determine the difference in energy consumption for language constructs in C#, whether they use more, the same, or less energy than other language constructs that are in the same group. This is done to give developers knowledge regarding the language constructs, so they can write code that is more energy-efficient.

For example, you can exchange a for loop with a while loop without changing the outcome of the code. In general, we aim to have equivalent outcomes when possible, but there are cases where this is not possible. For example, if we try to compare an `if` statement with an `exception`, it may not always be possible to switch one for the other. However, all the benchmarks in the same group will have similar functionality, meaning they can be switched for each other in some way.

When comparing these language constructs, various considerations must be made. To make benchmarks we follow five criteria from [31]:

1. Relevance, it is important to design the benchmark so it is relevant for its intended use.
2. Reproducibility, it is important to describe the test environment so the results are consistent between different testers.
3. Fairness, it is important the benchmarks compete without any unnecessary artificial constraints.
4. Verifiability, it is important to be able to verify the results are correct.
5. Usability, it is important to be able to use the benchmarks.

To follow these five criteria, we follow the setup and guidelines presented in [3, p.43-45] when writing benchmarks. We peer review the benchmarks to make sure the benchmarks have relevance, we run the benchmarks on a single computer sequentially to create reproducibility and fairness, we use the framework presented in [3] to have verifiability and usability in our

benchmarks. In addition to this, we calculate sanity checks in Chapter 3 to further verify our results.

2.2 Microbenchmarks vs Macrobenchmarks

To make benchmarks that are relevant for each part of the project, we first give a short definition of the concept of micro-and macro-benchmarks. A microbenchmark tests a little code, sometimes just a single operation. A macrobenchmark tests larger amounts of code, sometimes entire programs [32]. In [3] we created microbenchmarks for 316 different language constructs. An advantage of our microbenchmarks is that we test one specific language construct, keeping other variables constant. This gives a possibility to figure out what language constructs consume the least amount of energy compared to other language constructs which accomplish the same, or similar goals. A disadvantage of the microbenchmarks is that they may not be representative of an entire program, as a single language construct is a very small part of an entire program. An advantage of macrobenchmarks is that they can represent an entire program, they can therefore be used to measure the energy consumption of entire programs. A disadvantage of macrobenchmarks is that small changes such as a change in a single language construct are hard to measure.

2.3 Choice of Benchmarks

As mentioned, we continue with microbenchmarks from our previous project [3]. We do so because we have an improvement to our method for verifying our results by doing sanity checks. Furthermore, we show an improvement to our analysis of the results from our microbenchmarks. As the purpose is to improve the method for verifying and analyzing the results of microbenchmarks, we only need enough microbenchmarks to show the improved method. Therefore we take a step further with our research and look into larger benchmarks, where we use the results from the current microbenchmarks together with the results from our previous project, to test if the results from our microbenchmarks can be generalized to larger benchmarks. The larger benchmarks could be considered macrobenchmarks, however we will refer to them as larger benchmarks because macrobenchmarks often are considerably larger. We choose to use larger benchmarks

as opposed to macrobenchmarks, as it would take too much of our limited time to make all the changes needed in a macrobenchmark.

2.4 Measurement Approach

We choose to measure energy consumption and elapsed time for each of the benchmarks, so a developer can use the results no matter if the developer focuses on elapsed time or energy consumption, or a combination of the two. Furthermore, having both elapsed time and energy consumption, makes it possible to see if, and how, these correlate.

In [3] we found that Running Average Power Limit (RAPL) is an accurate measuring tool and has a finer granularity with regards to energy consumption compared to wall-socket/hardware measuring, therefore we will continue using the framework that was developed in [3].

2.5 Experiment Guidelines

We use guidelines to set up our benchmarks so it is possible to be consistent between benchmark developers.

In [3, p.37-39] we create guidelines for microbenchmarks on the basis of tips from [33], these tips being:

- 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 the standard deviation.

We follow these tips with the only difference being that instead of running the entire benchmark 10 times, we utilize a formula presented in [34] to calculate how many times we need to run the benchmark to get a significant result.

As we use the framework presented in [3], we use these guidelines automatically, except for saving the results inside the benchmark to a dummy variable.

Furthermore, with regards to the code setup, the same approach as in [3, p.38-39] is utilized.

In [3, p.38-39] we utilize a `for` loop around the measurements to create multiple iterations within the benchmark to make it run for longer. The framework created takes care of the other parts of the code setup for us, including initializing energy and time measurement, ending energy and time measurement, and normalizing the results.

The tip regarding saving results inside the benchmark to a dummy variable is not necessary for larger benchmarks, as large benchmarks not optimized away by the compiler. However, we still need to calculate the number of times the entire benchmark should be run to get a significant result.

2.6 Threats to Validity

We look into what threats to the validity of the results from our experiments have to be able to reflect on the results. We have divided the threats to validity into four categories. The categories are Construct validity, Internal Validity, External Validity, and Reliability. The categories are inspired from [35].

2.6.1 Construct validity

Construct validity refers to if we measure what we expect to be measuring, which in our case is the energy consumption and time elapsed for language constructs in our benchmarks. To ensure we measure this, we follow the tips in Section 2.5. Specifically, for the microbenchmarks we follow the tip to save the result inside the benchmarks so that the compiler does not compile the benchmarks away. Furthermore, we compare the results of the microbenchmarks to the result of running a microbenchmark with an empty loop, as if these are the same, we assume that the compiler has optimized the code in some way. We can do this because we found that empty loops are not optimized away by the compiler in [3]. If the benchmark is optimized away by the compiler we rewrite the benchmark until it is no longer optimized away by the compiler. For larger benchmarks, we do not need to consider these issues, as the larger benchmarks are large enough that the compiler can not optimize it away as that would alter the behavior of the benchmark. For the larger benchmarks, we only need to ensure that

the behavior stays the same after making the changes from the suggestions gathered by the microbenchmarks.

2.6.2 Internal Validity,

Internal validity refers to whether other factors than the variables used in the experiments influence the results. These threats include the temperature of the testing environment, System daemons, CPU frequency scaling, OS context-switching, and garbage collection. To mitigate the temperature being a factor, we monitor the temperature of the CPU during the experiments and only perform experiments when the CPU temperature is within the same temperature range or an established temperature interval. To mitigate OS context-switching we perform enough measurements to get significant results and shield the cores used for benchmark execution as described in [36].

2.6.3 External Validity

External validity refers to whether the results can be generalized. The main threat to validity here is whether the results from the microbenchmarks can be generalized to larger benchmarks. As one of the purposes of this project is to figure out whether the results found in [3] and from a few extra microbenchmarks can be generalized to larger benchmarks, we look into whether this threat to validity is of concern.

Furthermore, only testing a single large benchmark is not enough to figure out whether results can be generalized, which means we test multiple.

2.6.4 Reliability

Reliability refers to if the benchmarks are reproducible if you use the same methodology. There are a lot of areas that can affect reliability, such as random context switching, clock speed, and randomness with regards to the hardware states. We follow the same methodology as in [3] which includes the tips from [36] to 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 an 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.

2.7 Summary

The goal of this section is to determine the difference in energy consumption between language constructs that achieve the same or similar results in C#. To achieve this goal we add a few microbenchmarks to our list from our last project [3], where we examine the language constructs' energy consumption itself. With the results from the microbenchmarks we make larger benchmarks to examine the language constructs' effect on energy consumption for bigger programs. We measure both energy consumption and time elapsed, however, we focus on the energy consumption during this project. We choose to use RAPL as we have previous experience with it and we found that it is an accurate measurement tool. To make consistent, reliable, and reproducible results we create guidelines for our experiments, we follow the approach recommended in [33], however, we calculate how many times a benchmark should be run instead of following the recommended in [33]. We describe the different threats to validity that affect our benchmarks. These are divided into construct validity, internal validity, external validity, and reliability. In each category, we describe the threats and how to mitigate them.

Following the design, we create benchmarks and present the results in the following chapter.

Chapter 3

Benchmark Results

In this chapter, we look at the results from the benchmarks. All the benchmarks can be seen in our git repository [1]. We do this to gather knowledge that can be used to create suggestions that are used in *Energy Analyzer*.

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 6.0.101

Table 3.1: Specifications for the computer used in testing.

We use a server with the setup seen in Table 3.1, this is done to have consistency with regards to the results. This is the same server that was utilized in [3], meaning the results are directly comparable to the results from [3]. However, as computers are somewhat indeterministic, comparing results directly from this project to [3] should be done with care, and is not the sole reason for conclusive evidence.

Important to note is that the results are normalized to 1.000.000 iterations, i.e. the same as running the relevant benchmark 1.000.000 times like in [3].

3.1 Microbenchmarks

We start by creating and presenting microbenchmarks, as we want to use these results, together with the results from [3], for our larger benchmarks.

The procedure presenting our results for microbenchmarks mostly follows the same procedure we used in [3], however, we add an extra step with a sanity check to make sure our results are plausible.

Furthermore, we define a significant difference between two results as having a p-value of less than 0,05 [37] and having a difference of more than 2%, as previous work has found that the results can vary by up to around 1% [3, p.131-132]. If there is no significant difference, we can not say if there is an actual difference between the two results.

Procedure

The procedure for each section in microbenchmarks is:

1. Describe the benchmarks, what are the constructs used, and why are we testing them.
2. Show code for all of the benchmarks.
3. Show one or more plots of the results to give an overview of the results. In the plots, the dashed line shows the average, the solid line the median, the boxes contain the 95% interval within the results found, and the outer lines show the furthest outliers. In some cases, the average and median are on top of each other, and therefore hard to see. Furthermore, outliers in some cases do not happen so they are within the 95%.
4. Show tables with the numeric results for more detailed results.
5. Sanity check by taking the approximate power usage for a single core of the CPU and multiplying it with the elapsed time measured. The CPU has 6 cores and 12 threads and its Thermal Design Power (TDP) is 125 Watts [38]. We estimate the power consumption of a single core to be somewhere around 20 Watts, as 125 watts divided by 6 cores is around 20 watts per core. As our elapsed time is in milliseconds and our package energy is in microjoule, we multiply our elapsed time by 1000 to get to microseconds, and then multiply by 20 to get the power consumption in microjoule. We consider there to be no obvious errors, if the actual results show a deviation of less than a factor of 2 from the expected results.

3.1.1 Lambda Expressions

In this section, we look at the efficiency of different ways of creating lambda expressions. We do this to get insight into a part of `Invocation` that was partly overlooked in [3], and to show our improved methodology. Furthermore, this gives information that can be used to create suggestions for *Energy Analyzer*.

We create two groups for lambda expressions: Having the lambda expression outside the loop, meaning it is initialized before the main part of the benchmark is run, and having the lambda expression inside the loop, meaning it is initialized during every loop cycle.

This is done to get an understanding of the impact of moving an expression like that outside of a loop.

For each of these groups we create five benchmarks:

- `Lambda`,
- `LambdaAction`,
- `LambdaClosure`,
- `LambdaDelegate`, and
- `LambdaParameter`.

These five benchmarks give insight into several ways of creating lambda expressions.

Lambda Expressions Outside Loop

We start with the group of lambda expressions that are initialized outside the loop.

For this group, we look at the `Lambda` benchmark.

```
1 public class LambdaBenchmarks {
2     ...
3     [Benchmark("Lambda Expression", "Tests if using a func is
  ↪  better")]
4     public static ulong Lambda() {
5         ulong result = 0;
6         Func<ulong> test = () => 25;
```

```
7           for (ulong i = 0; i < LoopIterations; i++) {
8               result = test() + result + i;
9           }
10          return result;
11      }
12      ...
13  }
```

Listing 1: The `Lambda` method which tests a `Lambda` expression in the `LambdaBenchmarks` class.

In Listing 1 we can see the `Lambda` benchmark, in this benchmark we create a lambda expression that is assigned to a `Func`, meaning it has a return value. We call this lambda expression, add the return value to the `result` and add `i` to ensure that the benchmark is not compiled away.

The rest of the lambda expression benchmarks follow the same structure. In the `LambdaAction` we use `Action` instead of `Func`, in `LambdaClosure` we access a variable outside the lambda expression, in `LambdaDelegate` we use a `delegate` instead of `Func`, and in `LambdaParameter` we use a parameter instead of directly accessing a variable outside of the lambda expression.

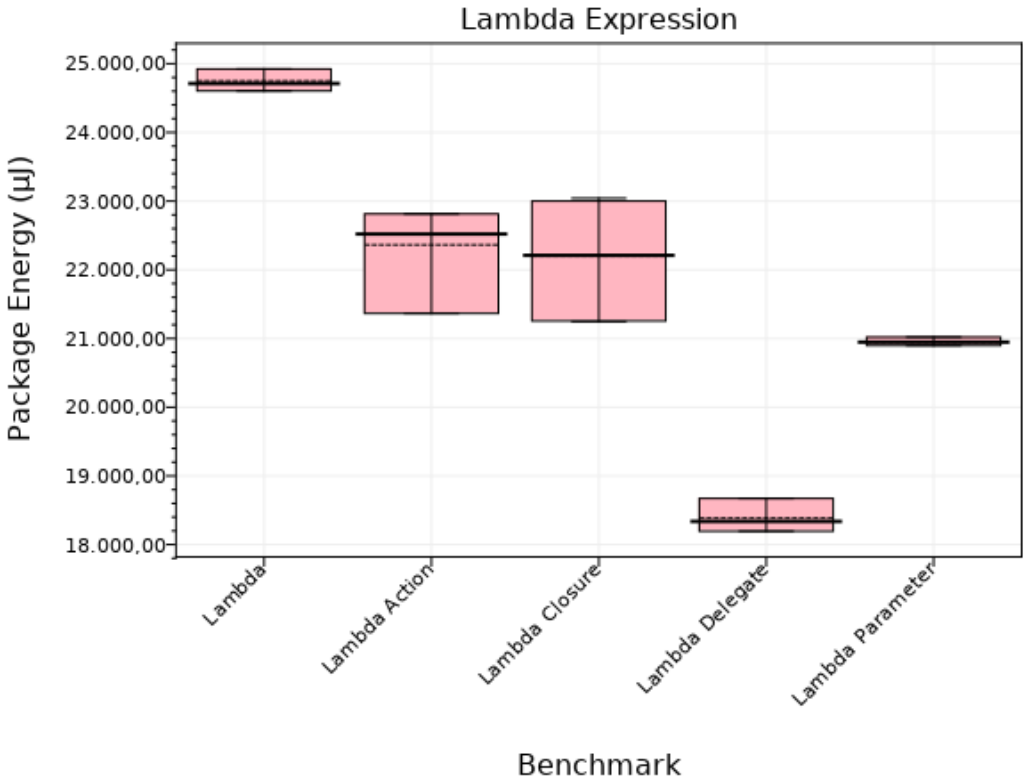


Figure 3.1: Energy Consumption for Lambda Expressions that are initialized outside the for loop.

In Figure 3.1 we can see the energy consumption for the benchmarks where the lambda expression is initialized outside the for loop in numeric form.

The y-axis does **not** start from zero to show a better view of the differences, and the span is around 7000µJ.

We can see that using a `delegate` is the most efficient while using an ordinary `Func` is the least efficient when looking at the energy efficiency. This is misleading, as the results differ a large amount from run to run for this group of benchmarks, which we will explore in Section 4.1.1.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Lambda	2,030996	24.740,375	1.139,994
Lambda Action	1,713301	22.352,621	962,034
Lambda Closure	1,713354	22.196,119	961,825
Lambda Delegate	1,468088	18.379,279	824,106
Lambda Parameter	1,713181	20.954,340	961,552

Table 3.2: Table showing the elapsed time and energy measurement for each Lambda Expression.

In Table 3.2 we can see the numeric results for each of the `Lambda Expression` benchmarks. As mentioned, the results are misleading, as the results differ a large amount from run to run.

We do a sanity check for all of them to see if the elapsed time fits with the amount of energy used, however, we only show the first one as they all show the same story.

$$1{,}713181ms * 1000 * 20W = 34.263{,}62\mu J \tag{3.1}$$

In Equation 3.1 we can see the sanity check for the `Lambda Parameter` benchmark, we can see that the energy consumption calculated is within a factor of 2 of the measured energy consumption, meaning there are no obvious errors in our measurements. The same is the case for all the other benchmark results.

The p-values for all of the results for this benchmark can be seen in Section A.1.1, however, because of the result variance, these are not conclusive.

Lambda Expressions Inside Loop

Next, we look at the group of lambda expressions that are initialized inside the loop.

For this group we look at the `InsideLoopLambda` benchmark.

```
1 public class LambdaBenchmarks {
2     ...
3     [Benchmark("Lambda Expression Inside Loop", "Tests if using
  ↪   a func is better inside the loop")]
4     public static ulong InsideLoopLambda() {
5         ulong result = 0;
6         for (ulong i = 0; i < LoopIterations; i++) {
```

```
7               ulong i1 = i;
8               Func<ulong> test = () => 25 + i1;
9               result = test() + result;
10          }
11          return result;
12      }
13      ...
14  }
```

Listing 2: The `InsideLoopLambda` method which tests a `Lambda` expression initilized inside the `for` loop in the `LambdaBenchmarks` class.

In Listing 2 we can see the `InsideLoopLambda` benchmark, this works in the same way as the `Lambda` benchmark shown in Listing 1, with the exception that the lambda expression is created inside the loop and the index variable is used inside the lambda expression. The same is the case for the rest of the `LambdaExpressionsInsideLoop` benchmarks, where they follow the same structure as the previous benchmarks, but with the lambda expressions defined inside the loop.

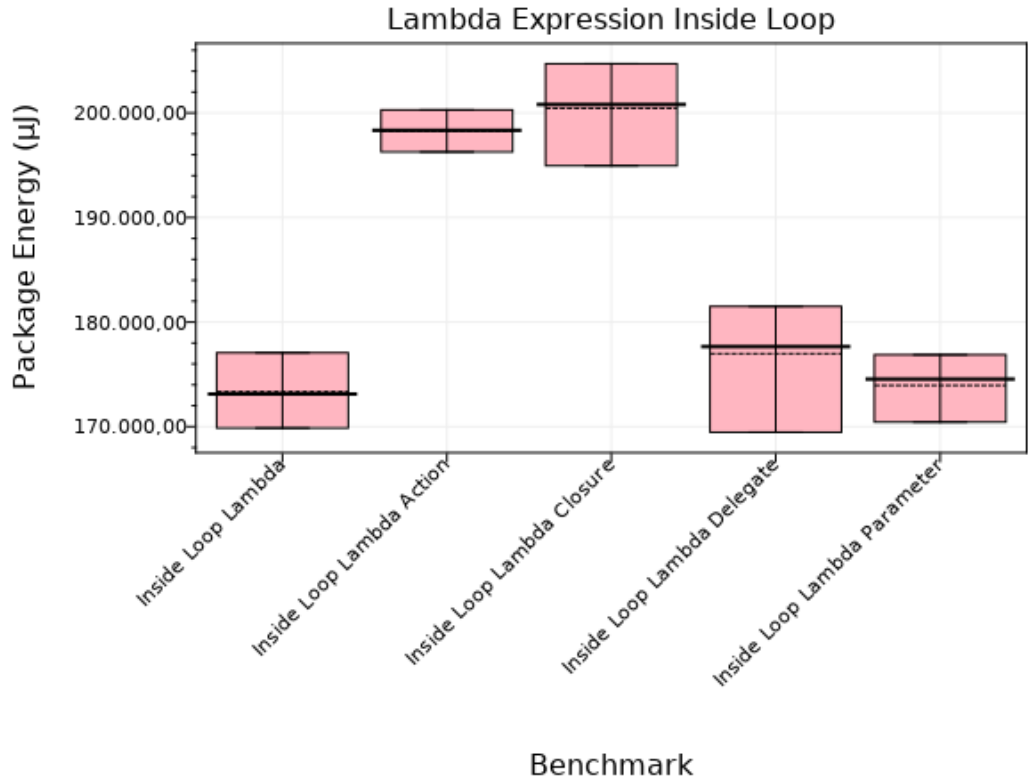


Figure 3.2: Energy Consumption for Lambda Expressions that are initialized inside the for loop

In Figure 3.2 we can see the energy consumption for the benchmarks where the lambda expression is initialized inside the for loop.

The y-axis does **not** start from zero to show a better view of the differences, and the span is around 180.000µJ.

We can see that using a `delegate`, ordinary `Func`, and `Func` with a parameter is the most efficient, while using an `Action` and `Func` with a surrounding variable is the least efficient when looking at the energy efficiency.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Inside Loop Lambda	12,232522	173.364,917	6.906,148
Inside Loop Lambda Action	14,114181	198.414,855	8.267,849
Inside Loop Lambda Closure	14,204120	200.548,161	8.286,308
Inside Loop Lambda Delegate	12,586698	177.045,568	7.428,007
Inside Loop Lambda Parameter	12,322410	173.911,275	7.251,037

Table 3.3: Table showing the elapsed time and energy measurement for each Lambda Expression Inside Loop.

In Table 3.2 we can see the numeric results for each of the `Lambda Expression` benchmarks. These results confirm what we concluded from the plot.

We do a sanity check for all of them to see if the elapsed time fits with the amount of energy used, however, we only show the first one as they all show the same story.

$$14{,}204120ms * 1000 * 20W = 284.082{,}4\mu J \tag{3.2}$$

In Equation 3.1 we can see the sanity check for the `Inside Loop Lambda Parameter` benchmark, we can see that the energy consumption calculated and the one measured are within a factor of 2 of each other, meaning there are no obvious errors in our measurements. The same is the case for all the other benchmark results.

The p-values for all of the results for this benchmark can be seen in Section A.1.2.

Findings

The findings in this section can be used to create suggestions that are used in *Energy Analyzer*, so developers can improve their programs with regards to energy consumption. Seeing if some of these results can be generalized to a larger benchmark makes it possible to create more precise suggestions for *Energy Analyzer*. To summarize, the results are as follows:

- The energy consumption of lambda expressions is inconsistent when initializing the lambda expressions outside the `for` loop.
- Using the `Action` construct and using closure with a lambda expression is less efficient than not doing that when initializing the lambda expressions inside the `for` loop.

3.1.2 Exceptions

We have created two groups for exceptions: Creating an exception and throwing/catching an exception. We have done this to get an understanding of the individual parts of exceptions, which is unlike what we did in [3]. This can also give more information that can be used to create more precise suggestions for *Energy Analyzer*.

Creating Exceptions

First we look at the group that creates exceptions, which contains three benchmarks.

- `CreateArgumentException`,
- `CreateDivideByZeroException`, and
- `CreateException`.

We create these to get an understanding of how much energy is consumed to create different types of exceptions.

We start by looking at the benchmark `Create Argument Exception`.

```
1  public class ExceptionBenchmarks {
2      ...
3      [Benchmark("Exception Creation", "Tests how much it costs to
   ↪   create an ArgumentException")]
4      public static Exception CreateArgumentException() {
5          Exception result = new Exception();
6          for (ulong i = 0; i < LoopIterations; i++) {
7              result = new ArgumentException();
8          }
9          return result;
10     }
11     ...
12 }
```

Listing 3: The `Create Argument Exception` method which create arguments exceptions in the `ExceptionBenchmarks` class.

In Listing 3 we see how we have created the benchmarks for creating `Argument Exceptions`. We create a new `Argument Exception` for as many times as the variable `LoopIterations` has been set to, after which the results are returned. The same is the case for the other benchmarks in this group, with the only difference being what type of `exception` is created.

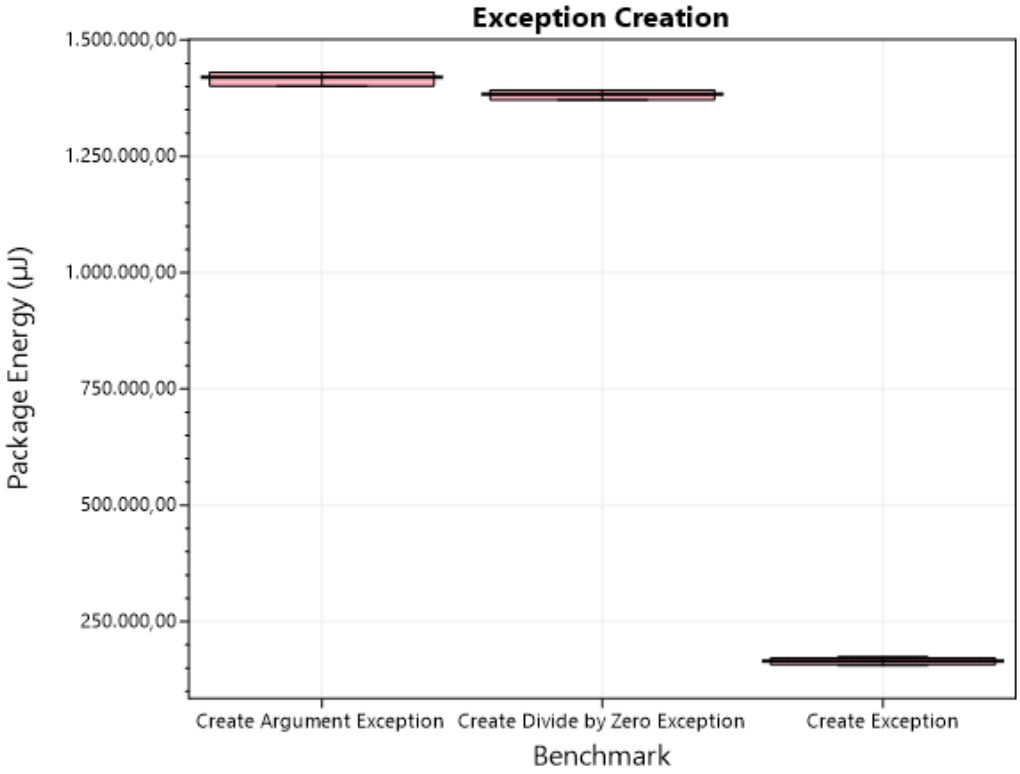


Figure 3.3: Energy Consumption for creating exceptions of different types.

In Figure 3.3 we can see the energy consumption for creating exceptions of types: `ArgumentException`, `DivideByZeroException`, and `Exception`. The y-axis does **not** start from 0 and spans around 15.000.000µJ. We see that creating an `Exception` consumes less energy than creating `Create Argument Exceptions` and `Divide By Zero Exceptions`.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Create Argument Exception	106,050686	1.418.835,983	77.073,708
Create Divide by Zero Exception	102,413879	1.383.990,746	73.907,242
Create Exception	11,594180	165.185,133	22.430,923

Table 3.4: Table showing the elapsed time and energy measurement for each Exception Creation.

In Table 3.4 we see the results for creating `Exceptions` in numeric form. We can see the average results for our three result types, Time Elapsed, Package Energy Consumption, and DRAM Energy Consumption. To verify

the results are plausible we make a sanity check on our package energy results.

$$11{,}59418ms * 1000 * 20W = 231.883{,}6\mu J \tag{3.3}$$

In Equation 3.3 we see the result for our sanity check for `Create Exception`, which is within a factor of 2 to our measured result, therefore we conclude that there are no obvious errors in our measurement. The rest of the benchmarks show the same.

In Table 3.4 we see that creating an `Exception` is more than 9 times faster on average than creating an `ArgumentException` or a `DivideByZeroException`. We can also see that creating an `Exception` consumes 8 times less package energy than creating an `ArgumentException` or a `DivideByZeroException`. The p-values for all the results are less than 0,05, meaning that the difference between all the exceptions are significant.

Throwing and Catching Exceptions

In this section we look at the group that throws and catches exceptions, which contains 12 benchmarks.

- `CachedArgumentException`,
- `CachedDivideByZeroException`,
- `CachedException`,
- `NewArgumentException`,
- `NewDivideByZeroException`,
- `NewException`,
- `ThrowCachedArgumentException`,
- `ThrowCachedDivideByZeroException`,
- `ThrowCachedException`,
- `ThrowNewArgumentException`,
- `ThrowNewDivideByZeroException`, and
- `ThrowNewException`.

The difference between the benchmarks that does not have `Throw` in their name compared with those that have `Throw` in their name, is that the ones that have `Throw` in their name have an addition occurring in the `catch` block. The benchmarks that have `New` in their name, creates a new `exception` each time an `exception` is thrown, while the ones without `New` in their name, uses an `exception` created outside the loop, i.e. a cached `exception`. We create and test these benchmarks to get an understanding of how much energy is consumed throwing and catching different types of `exceptions`. We start by looking at the benchmark `Cached Exception`.

```
1  public class ExceptionBenchmarks {
2      ...
3      [Benchmark("Exception", "Tests try-catch where a cached
   ↪   CachedArgumentException is thrown")]
4      public static ulong CachedArgumentException() {
5          ArgumentException argumentException = new
           ↪  ArgumentException();
6          for (ulong i = 0; i < LoopIterations; i++) {
7              try {
8                  throw argumentException;
9              }
10             catch { }
11         }
12         return LoopIterations;
13     }
14     ...
15 }
```

Listing 4: The `CachedArgumentException` method which tests a try catch throwing a cached exception in the `ExceptionBenchmarks` class.

In Listing 4 we see how we have created the benchmark for throwing and catching a cached `ArgumentException`. We throw and catch an `ArgumentException` as many times as the variable `LoopIterations` has been set to, after which the `LoopIterations` is returned. The `CachedDivideByZeroException` and `CachedException` benchmarks are almost similar, except they use a `DivideByZeroException` and an `Exception` respectively.

```
public class ExceptionBenchmarks {
    ...
    [Benchmark("Exception", "Tests try-catch with a new
    ↪ exception thrown")]
    public static ulong NewArgumentException() {
        for (ulong i = 0; i < LoopIterations; i++) {
            try {
                throw new ArgumentException();
            }
            catch { }
        }
        return LoopIterations;
    }
    ...
}
```

Listing 5: The `NewArgumentException` method which tests a try catch throwing a new ArgumentException in the `ExceptionBenchmarks` class.

In Listing 5 we see the benchmark for throwing and catching a new `ArgumentException`. It is similar to the `CachedArgumentException` benchmark, but instead of throwing the same `ArgumentException`, we create a new in each iteration of the for loop. The `NewDivideByZeroException` and `NewException` benchmarks are almost similar, except they use a `DivideByZeroException` and an `Exception` respectively.

```
public class ExceptionBenchmarks {
    ...
    [Benchmark("Exception", "Tests try-catch a cached exception
    ↪ is thrown and addition occurs")]
    public static ulong ThrowCachedArgumentException() {
        ArgumentException argumentException = new
        ↪ ArgumentException();
        ulong result = 0;
        for (ulong i = 0; i < LoopIterations; i++) {
            try {
                throw argumentException;
```

```
            }
            catch {
                result += i;
            }
        }
        return result;
    }
    ...
}
```

Listing 6: The `ThrowCachedArgumentException` method, which tests a try catch throwing a cached exception and an addition is made when the exception is caught, in the `ExceptionBenchmarks` class.

In Listing 6 we see the benchmark for throwing and catching a cached `ArgumentException`, and then making an addition when the exception is caught. It is very similar to the `CachedArgumentException` benchmark, however, we make an addition in the catch block. The `ThrowCachedDivideByZeroException` and `ThrowCachedException` benchmarks are almost similar, except they use a `DivideByZeroException` and an `Exception` respectively.

```
public class ExceptionBenchmarks {
    ...
    [Benchmark("Exception", "Tests try-catch with a new
↪   exception thrown and addition occurs")]
    public static ulong ThrowNewArgumentException() {
        ulong result = 0;
        for (ulong i = 0; i < LoopIterations; i++) {
            try {
                throw new ArgumentException();
            }
            catch {
                result += i;
            }
        }
        return result;
    }
    ...
}
```

Listing 7: The `ThrowNewArgumentException` method, which tests a try catch throwing a new exception and an addition is made when the exception is caught, in the `ExceptionBenchmarks` class.

In Listing 7 we see the benchmark for throwing and catching a new `ArgumentException`, and then making an addition when the exception is caught. It is similar to the other benchmarks, as it throws a new `ArgumentException` like the `NewArgumentException` benchmark, while making an addition in the catch block like the `ThrowCachedArgumentException` benchmark. The `ThrowNewDivideByZeroException` and `ThrownewException` benchmarks are almost similar, except they use a `DivideByZeroException` and an `Exception` respectively.

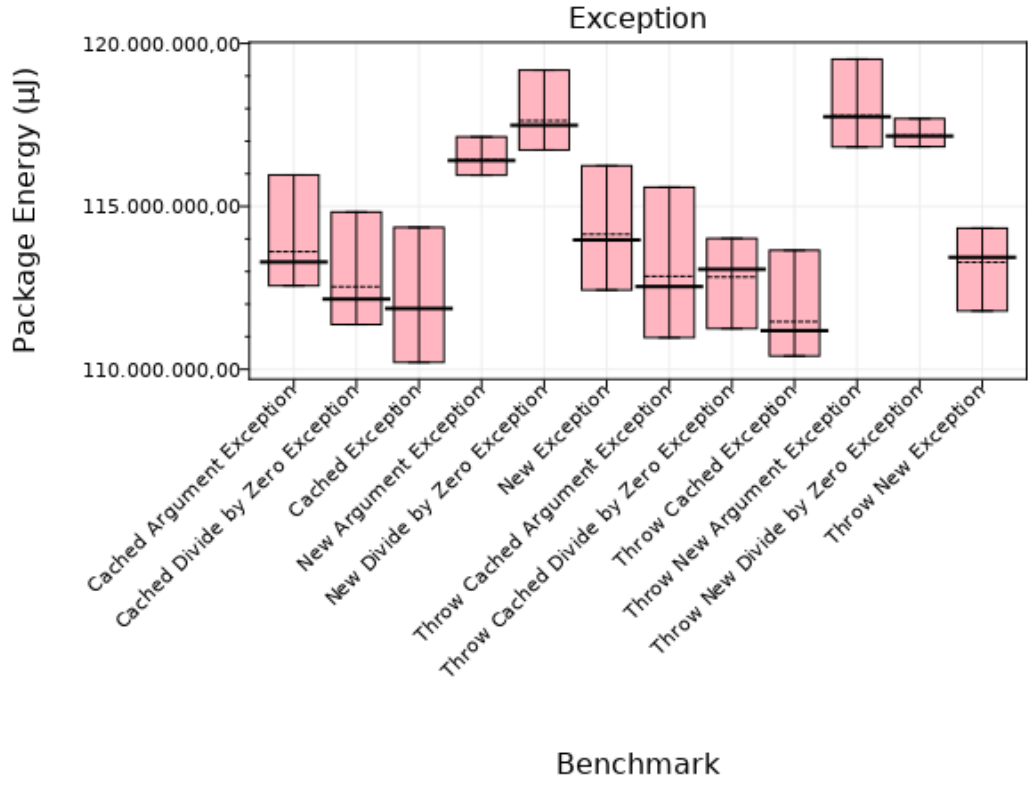


Figure 3.4: Energy Consumption for throwing and catching exceptions.

In Figure 3.4 we can see the energy consumption for throwing and catching exceptions that are created during the throwing or created before the

main part of the benchmark. The y-axis does **not** start from 0 and spans around 10.000.000µJ. We see that there is a lot of uncertainty with the results as they vary a lot. However, we see that the benchmarks using a cached exception consistently consumes less energy than the benchmarks creating new exceptions and that using an `Exception` is more energy-efficient than using either an `ArgumentException` or a `DivideByZeroException`.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Cached Argument Exception	8.398,393333	113.597.356,623	4.732.055,664
Cached Divide by Zero Exception	8.396,791178	112.520.737,712	4.721.295,166
Cached Exception	8.302,432658	111.814.550,781	4.667.028,809
New Argument Exception	8.737,015381	116.530.510,254	4.909.177,246
New Divide by Zero Exception	8.751,158203	117.557.790,527	4.926.459,961
New Exception	8.409,609222	114.159.925,842	4.729.153,442
Throw Cached Argument Exception	8.350,120117	112.799.786,377	4.689.979,248
Throw Cached Divide by Zero Exception	8.370,584961	112.777.062,988	4.705.329,590
Throw Cached Exception	8.279,091797	111.499.955,611	4.651.859,908
Throw New Argument Exception	8.765,054687	117.695.327,148	4.927.653,809
Throw New Divide by Zero Exception	8.765,690918	117.252.763,672	4.928.398,438
Throw New Exception	8.412,026855	113.348.371,582	4.729.018,555

Table 3.5: Table showing the elapsed time and energy measurement for each Exception.

In Table 3.5 we see the results for throwing and catching exceptions in numeric form. We see the result for Time Elapsed, Package Energy Consumption, and DRAM Energy Consumption. To verify the results are plausible we make a sanity check on our package energy.

$$8406{,}333984ms * 1000 * 20W = 168.126.679{,}68\mu J \tag{3.4}$$

Equation 3.4 shows the result for our sanity check for the `Throw Cached Exception` benchmark. The calculated energy consumption and the measured energy consumption are within a factor of 2, this result shows that our measurement does not have obvious errors. The same has been done for the rest of the results, showing the same. The p-values which can be seen in A.1.3, are all less than 0,05 between the cached exceptions and the new exceptions benchmarks, and between `Exception` benchmark and the `ArgumentException` or the `DivideByZeroException` benchmarks. The benchmarks which uses a cached exception differ more than 2% from the same type of benchmarks using a new exception, except for `Throw Cached Exception` and `Throw New Exception`. Moreover the benchmarks which are not cached and uses `Exceptions` differs in package energy with more than 2% from the similar benchmarks using an `ArgumentException` or a

`DivideByZeroException`. We conclude that the benchmarks which have a p-value less than 0,05 between them and differ with more than 2% in package energy consume different amounts of energy.

Findings

The findings in this section can be used as a starting point for creating suggestions that can be used in *Energy Analyzer* that will make it possible for developers to improve their programs with regards to energy consumption without much work. Seeing if some of these results can be generalized is another step in making formal suggestions for *Energy Analyzer*. To summarize, the results are as follows:

- Creating an `Exception` consumes 8 times less package energy than creating an `ArgumentException` or a `DivideByZeroException`.
- Throwing and catching an `Exception` is more energy efficient than using an `ArgumentException` or a `DivideByZeroException`.
- Throwing and catching a cached exception consumes less energy than throwing and catching a newly created exception when using `ArgumentException` or `DivideByZeroException`.

3.2 Larger benchmarks

We now create and present the larger benchmark results, as this shows how well the results from the microbenchmarks can be generalized. The results from the larger benchmarks, together with the analysis in Chapter 4 serve to create formal suggestions for *Energy Analyzer*. The procedure for presenting our larger benchmarks takes inspiration from the way we present microbenchmarks in Section 3.1.

Procedure

The procedure for each benchmark is:

1. Describe the benchmark, and what language constructs have been used.
2. Show parts of the code that changed.

3. Show one or more plots of the results to give an overview of the results.

4. Show tables with the numeric results for more detailed results.

5. Sanity check by taking the approximate power usage for a single core of the CPU and multiplying it with the elapsed time measured, like for the microbenchmarks.

3.2.1 Benchmark Overview and Choices

To get an overview of the changes made to the larger benchmarks, we create a list of changes, seen in Section A.2. This list is used as a guideline for the changes done in all of the large benchmarks tested, the list is based upon research and reflections from [3] and the results from the microbenchmarks from Section 3.1.

Furthermore, it is important to know which benchmarks we use, to be able to replicate the results.

To select our benchmarks we look at using the publicly available benchmarks from *RosettaCode* [39] and the *Computer Language Benchmarks Game*[40].

We find *RosettaCode* focus more on having a wide selection of benchmarks whereas the *Computer Language Benchmarks Game* focus more on singular algorithms and on creating different variations of these algorithms.

We decide to use various benchmarks from *RosettaCode* as this allows for a wide selection of benchmarks to cover more areas like reflection, lambda expressions, string concatenation, etc. This serves to better reflect our microbenchmarks in the more general setting of the large benchmarks.

We choose the following nine benchmarks:

- 2048
- 21 game
- 4 rings or 4 squares puzzle
- 99 bottles of beer
- Determine if a string has all the same characters
- Dijkstra's algorithm
- Happy numbers

- Introspection
- World Cup group stage

We choose these benchmarks to get a comprehensive suite of benchmarks that can test all of the groups of changes seen in Section A.2. Important to note is that some individual suggestions within each group are not used, as benchmarks, where these would be applicable, could not be found without the benchmark being specific for that individual suggestion. As we want general benchmarks, we choose not to use benchmarks where only one specific suggestion is used. The specifics with regards to what groups of changes have been made to each benchmark can be seen in their respective section.

Furthermore, instead of doing all of the changes every time, we do batches of changes to figure out which changes have the most impact on every benchmark. These batches can be seen in Section A.2.1.

This also serves to show if any of the changes affect the benchmarks in a way that is contrary to what we expect.

We compare the results of each of these to a benchmark where no changes have been made and to a benchmark where all of the changes have been made. In cases where changes overlap, we make changes that would have the largest impact according to the microbenchmark results.

The p-values for all of these benchmarks can be seen in Section A.3.

3.2.2 2048

The benchmark [41] is an implementation of the game `2048`, which is a sliding puzzle game where blocks with the same numbers are combined to grow the value of a block. There are several rules to this game which can be seen in [41].

The `2048` benchmark is used to show whether the batches from Section A.2.1, with regards to `Datatypes`, `Selection`, `Loops`, and `Objects`, have the desired effect when generalizing to larger benchmarks.

The change with regards to `Datatypes` is changing the `int` variables to `uint`. Furthermore, we change the method of `string` concatenation from using the concatenation operator (+) to using `StringBuilder`.

The change with regards to `Selection` is changing `if` statements to `switch` statements where possible.

The change with regards to `Loops` is changing iterating through an `array` from a `for` loop to a `foreach` loop.

Lastly, the change with regards to `Objects`, is changing `class` to `struct`.

Code Changes

To make it possible to run the benchmark as part of our framework [3], it is necessary to make changes besides the ones needed to test the language constructs.

Specifically, we move the contents of the constructor to the `InitializeBoard` method, to make it possible to reset the board between each iteration.

```
1  public class Default {
2      internal class G2048 {
3          public void InitializeBoard() {
4              _isDone = false;
5              _isWon = false;
6              _isMoved = true;
7              _inputs = new Queue<char>(Enumerable.Repeat('w',
               ↪  100));
8              _score = 0;
9              for (int y = 0; y < 4; y++) {
10                 for (int x = 0; x < 4; x++) {
11                     _board[x, y] = new Tile();
12                 }
13             }
14         }
15     }
16 }
```

Listing 8: Change to Default implementation of 2048, `InitializeBoard` method.

In Listing 8 we can see how the `InitializeBoard` method looks after the change. Important to note is that another change made is that a field called `_inputs` is initialized so that we do not need to manually make inputs to the game when running the benchmark. This is used in the `WaitKey` method.

```
1  public class Default {
2      internal class G2048 {
3          private void WaitKey() {
```

```
4               _isMoved = false;
5               if (_inputs.Count == 0) {
6                   _isDone = true;
7                   return;
8               }
9               switch (_inputs.Dequeue()) {
10                  case 'W':
11                      Move(MoveDirection.Up);
12                      break;
13                  case 'A':
14                      Move(MoveDirection.Left);
15                      break;
16                  ...
17              }
18              ...
19          }
20      }
21  }
```

Listing 9: Change to Default implementation of 2048, `WaitKey` method.

In Listing 9 we can see that the `WaitKey` method dequeues the `_inputs` field to simulate moving the blocks in 2048.

With regards to the changes to datatypes, we naively change all occurrences of `int` to `uint`. In cases where `int` is necessary, we cast from `uint` to `int`. We also naively change occurrences of the string concatenation operator (+) to `StringBuilder`, which is then used together with a `.ToString()` call when necessary.

```
1   public class DataType {
2       internal class G2048 {
3           private void DrawBoard() {
4               Console.Clear();
5               StringBuilder sb = new StringBuilder("Score: ");
6               sb.Append(_score);
7               sb.Append("\n");
8               Console.WriteLine(sb.ToString());
9               for (uint y = 0; y < 4; y++) {
```

```
10                  ...
11              }
12              ...
13          }
14      }
15  }
```

Listing 10: Example of datatype change to 2048, `DrawBoard` method.

In Listing 10 we can see an example of datatype changes to 2048. On line 5 to 8 we can see how a `StringBuilder` is created and written to `Console`. On line 9 we can see that `uint` is used instead of `int` in the `for` loop.

With regards to the changes to selection, the approach is the same, where we naively change `if/else` occurrences with `switch` if possible.

```
1  public class Selection {
2      internal class G2048 {
3          private void MoveVertically(int x, int y, int d) {
4              ...
5              switch (d) {
6                  case > 0: {
7                      if (y + d < 3) {
8                          MoveVertically(x, y + d, 1);
9                      }
10                     break;
11                 }
12                 default: {
13                     if (y + d > 0) {
14                         MoveVertically(x, y + d, -1);
15                     }
16                     break;
17                 }
18             }
19         }
20     }
21 }
```

Listing 11: Example of selection change to 2048, `MoveVertically` method.

In Listing 11 we can see an example of selection change to 2048. On line 5 to 18, we can see how a `switch` is used instead of an `if/else` where we check if the variable `d` is above 0.

With regards to the changes to loops, we look for places where we iterate through a collection and change that to using a `foreach` loop.

```
1  public class Loops {
2      internal class G2048 {
3          private void WaitKey() {
4              ...
5              foreach (Tile tile in _board) {
6                  tile.IsBlocked = false;
7              }
8          }
9      }
10 }
```

Listing 12: Example of loop change to 2048, `WaitKey` method.

In Listing 12 we can see an example of loop change to 2048. On line 5 to 7 we can see how a `foreach` loop is used to change the `IsBlocked` field on each tile in `_board` to false.

Lastly, with regards to objects, we naively change `class` to `struct`.

```
1  public class Object {
2      internal struct G2048 {
3          public G2048() {
4              _isDone = default;
5              _isWon = default;
6              _isMoved = default;
7              _score = default;
8              _inputs = null;
9          }
10     }
11 }
```

Listing 13: Example of objects change to 2048, `G2048 struct`.

In Listing 13 we can see how the `G2048` `class` is changed to a `struct`. This also means that a default constructor needs to be implemented, as this is necessary when using a `struct`. The default constructor can be seen on lines 3 to 9, this constructor initializes all variables in the struct to the default value. Their initial values are still set in the `InitializeBoard` method, so the values set in the constructor are not used.

The implementation of all of these changes in one class is also created, to see what the effect is when all of the changes are made together.

Results

We compare the results of the different implementations of `2048` to see if the results can be generalized when implemented naively, or whether we need to analyze to figure out how the language constructs should be used in general implementations to achieve energy efficiency improvements.

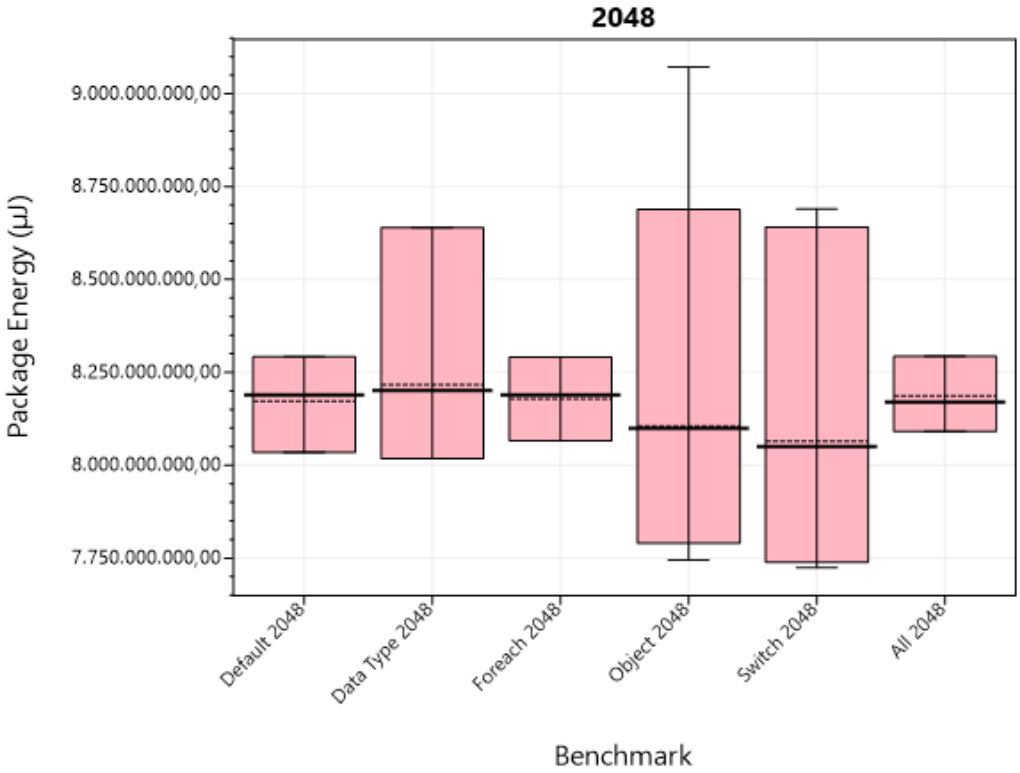


Figure 3.5: Energy consumption of each implementation of `2048`.

In Figure 3.5 we can see the energy consumption for the different im-

plementations of 2048. The y-axis does **not** start from zero and the span is around 1.500.000.000µJ. Here we can see that there are no significant differences between any of the implementations.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Default 2048	281.926	8.171.909.912	157.448.975	N/A
Data Type 2048	283.261	8.216.815.682	158.238.407	0,55%
Foreach 2048	282.802	8.178.229.403	158.022.372	0,08%
Object 2048	281.644	8.104.721.753	157.182.399	-0,82%
Switch 2048	280.628	8.064.600.838	156.666.319	-1,31%
All 2048	283.727	8.186.948.568	158.587.674	0,18%

Table 3.6: Table showing the elapsed time and energy measurement for each 2048, the difference from default is with regards to package energy.

Looking at Table 3.6, we can see that all of the results with regards to package energy are within 2% of the default implementation, meaning there are no significant differences when naively implementing these changes to 2048.

To verify that the results are plausible, we make a sanity check on our package energy.

$$281.643{,}503673ms * 1000 * 20W = 5.632.870.073{,}46\mu J \tag{3.5}$$

Equation 3.5 shows the result of our sanity check for the 2048 object benchmark. The calculated energy consumption is within a factor of 2, meaning that there are no obvious errors to the benchmark. The same is the case for the other benchmarks.

Because there are no significant differences between the different implementations, we analyze to understand how the language constructs should be used to get the best energy efficiency in 2048. Furthermore, we see if any suggestions can be generalized so that they always decrease energy consumption in Section 4.2.

3.2.3 21 Game

The benchmark [42] is an implementation of a 2-player game where each player, in turn, adds either 1, 2, or 3 to the running total which starts at 0. The player who causes the running total to hit exactly 21 is the winner, the specifics of which can be seen in [42].

The 21 game benchmark is used to show whether the batches from Section A.2.1, with regards to Datatypes, Selection, and Exceptions have the desired effect when generalizing to larger benchmarks.

The change with regards to Datatypes is changing the int variables to uint. Furthermore, we change string interpolation to StringBuilder.

The change with regards to Selection is changing if statements to switch statements when possible.

The change with regards to Exceptions is replacing the exceptions with if statements when possible.

Code Changes

No changes to the code are necessary to make the default implementation of 21 work.

The changes to datatypes are the same naive changes done as in 2048, seen in Section 3.2.2, the same is the case with the selection changes.

With regards to exceptions, we remove all occurrences of try-catch.

```
1  public class Exceptions {
2      public static void PlayGame() {
3          ...
4          while (playAnother) {
5              Console.WriteLine($"Now playing: {currentPlayer}")
6              // Removed try-catch block here
7              ...
8              if (roundChoice != 1 && roundChoice != 2 &&
                   ↪ roundChoice != 3) {
9                  // Removed exception throw here
10                 Console.WriteLine("Invalid choice! Choose from
                       ↪ numbers: 1, 2, 3.");
11                 continue;
12             }
13             ...
14         }
15     }
16 }
```

Listing 14: Example of exception change to 21, PlayGame method.

In Listing 14 we can see that a `try-catch` block has been removed and an `exception` throw has been removed, no functionality has changed as a result from this.

The implementation of all of these changes in one class is also created, as in 2048, to see what the effects are of having all of these changes in one benchmark.

Results

We compare the results of the different implementations of `21` to see what effects the changes have to the energy consumption of `21`.

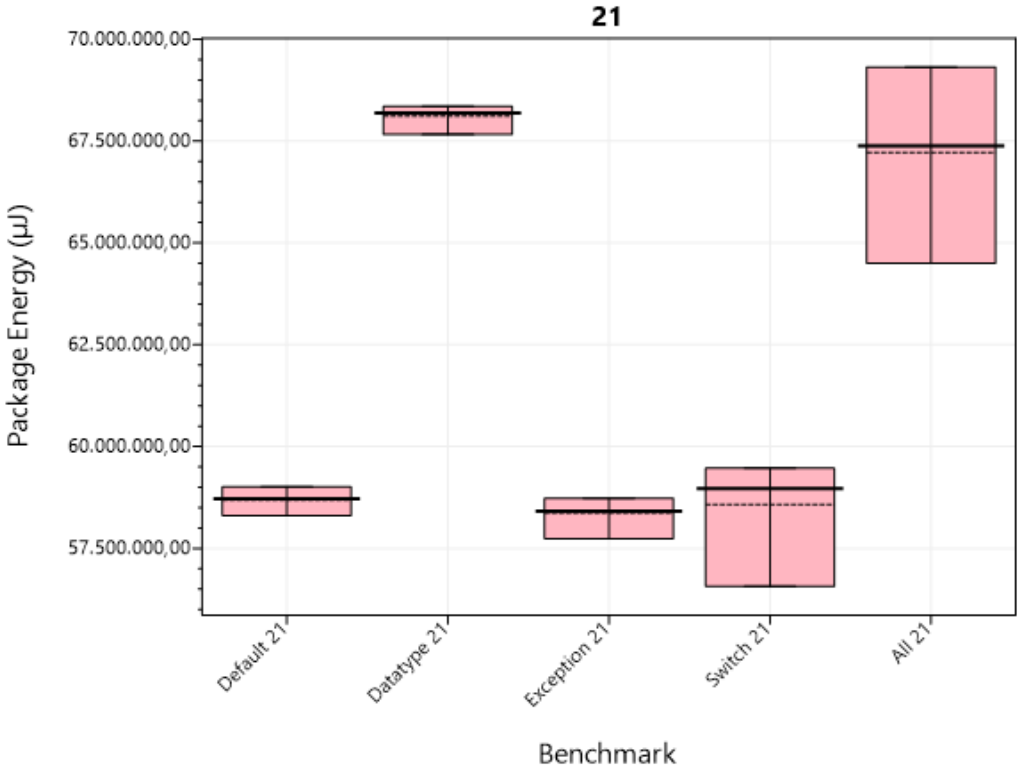


Figure 3.6: Energy consumption of each implementation of `21`.

In Figure 3.6 we can see the energy consumption of the different implementations of `21`. The y-axis does **not** start from zero and the span is around 15.000.000µJ. Here we can see that the changes to datatype increases the energy consumption, which is surprising.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Default 21	4.469	58.685.176	2.493.352	N/A
Datatype 21	5.172	68.117.483	2.894.853	16,07%
Exception 21	4.467	58.369.766	2.490.455	-0,54%
Switch 21	4.485	58.570.674	2.501.339	-0,20%
All 21	5.168	67.220.656	2.892.854	14,54%

Table 3.7: Table showing the elapsed time and energy measurement for each 21, the difference from default is with regards to package energy.

In Table 3.7, we see the same as in the plot, that there is a significant difference between the datatype (and all) implementation compared to the default implementation, while all other implementations have no significant difference.

To verify that the results are plausible, we calculate a sanity check on our package energy.

$$4.484{,}737200ms * 1000 * 20W = 89.694.744\mu J \tag{3.6}$$

In Equation 3.6 we can see the result of our sanity check for the `21 switch` benchmark. The calculated energy consumption is within a factor of 2, meaning that there are no obvious errors in the benchmark. The same is the case for the other benchmarks.

Because the results show that datatype changes increase the energy consumption and that no other change has a significant difference, we do analysis to figure out what changes to the `21` benchmark can be done to get the best energy efficiency, and what suggestions should be given to the developer if any in Section 4.2.2.

3.2.4 4-Rings or 4-Squares Puzzle

The benchmark [43] is an implementation of solutions to a puzzle where letters in 4 squares should be replaced with digits within a range of numbers such that the sum of the letters inside of each square add up to the same sum [43].

The 4-Rings or 4-Squares Puzzle benchmark is used to show whether the batches from Section A.2.1, with regards to `Datatypes`, and `LINQ`, has the desired effect when generalizing to larger benchmarks.

The changes with regards to `Datatypes` is changing `int` variables are changed to `uint` variables. Furthermore, we change the method of `string` concatenation when using `Console.WriteLine`, to using `StringBuilder`.

The change with regards to Language Integrated Query (LINQ) is removing and replacing any occurrences of LINQ where possible.

Code Changes

No changes to the code are necessary to make the default implementation of 4-Rings or 4-Squares Puzzle work.

The changes to datatypes are the same naive changes done in 2048, seen in Section 3.2.2. With regards to LINQ, we remove and replace any occurrence of LINQ.

```
1  public class LINQ {
2      private static bool NotValid(bool unique, int needle, params
       ↪  int[] haystack) {
3          if (unique) {
4              foreach (int i in haystack) {
5                  if (i == needle) {
6                      return true;
7                  }
8              }
9          }
10         return false;
11     }
12 }
```

Listing 15: Example of removing and replacing LINQ in the 4-Rings or 4-Squares Puzzle benchmark, `NotValid` method.

In Listing 15 we can see how a `.Any` LINQ method has been replaced with a `foreach` loop that checks for the same condition.

An implementation of the changes in both of these categories is also created, as in the previous benchmarks.

Results

We compare the results of the different implementations of the 4-Rings or 4-Squares Puzzle to see what effects the changes have on the energy consumption of this benchmark.

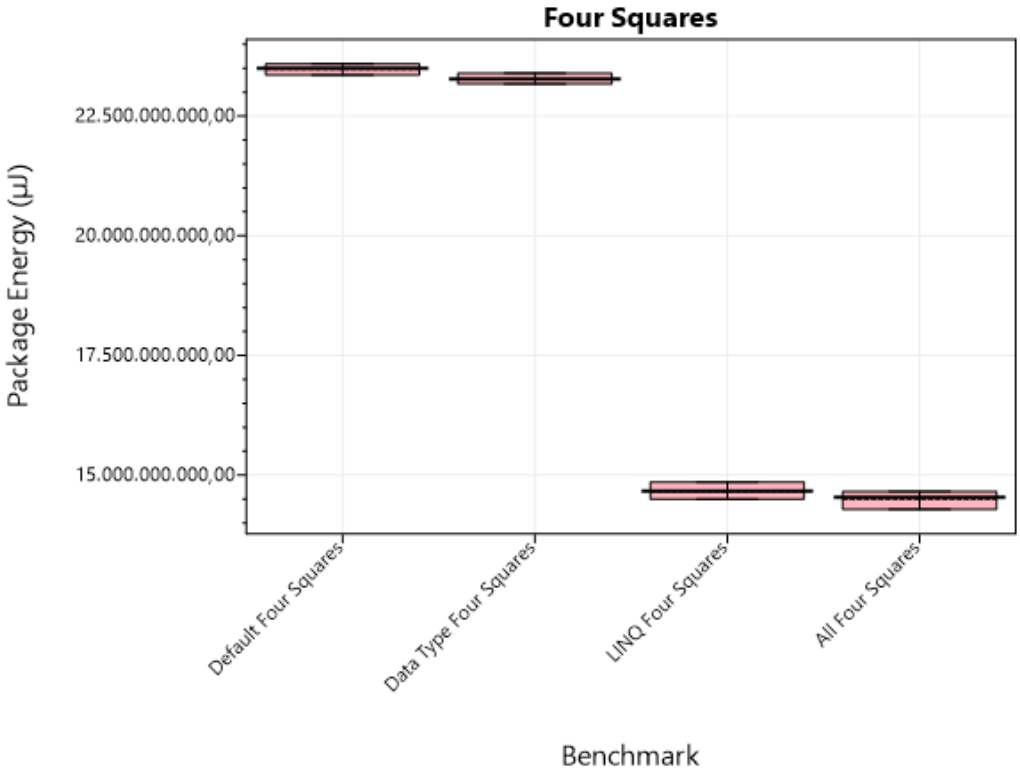


Figure 3.7: Energy consumption of each implementation of the 4-Rings or 4-Squares Puzzle.

In Figure 3.7, we can see the energy consumption of the different implementations of the 4-Rings or 4-Squares Puzzle. The y-axis does **not** start from zero and the span is around 10.000.000.000µJ. Here we can see that the changes to `LINQ` decreases the energy consumption significantly.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Default Four Squares	1.784.827	23.477.282.118	999.845.486	N/A
Data Type Four Squares	1.762.870	23.264.401.476	981.884.549	-0,91%
LINQ Four Squares	1.115.392	14.662.122.396	621.370.226	-37,55%
All Four Squares	1.106.992	14.495.177.083	618.932.726	-38,26%

Table 3.8: Table showing the elapsed time and energy measurement for each Four Squares, the difference from default is with regards to package energy.

In Table 3.8 we can see the same as in the plot, however here it is more clear to see that the `datatype` changes have no significant impact on the energy consumption, as it is within 2% of the default implementation.

To verify that the results are plausible, we do a sanity check on our package energy.

$$1.115.392{,}361111ms * 1000 * 20W = 22.307.847.222,22\mu J \tag{3.7}$$

In Equation 3.7 we can see that the sanity check shows that there are no obvious errors to the result of the `LINQ` benchmark, as the result is within a factor of 2 to the actual result. The same is the case for the other benchmarks.

As the results show what we would expect given the results from our microbenchmarks, we do not do any analysis of these results.

3.2.5 99 Bottles of Beer

The benchmark [44] is an implementation of displaying the lyrics to the "99 Bottles of Beer on the Wall" song. Specifically, we look at the "Flexible" implementation for C#.

The 99 Bottles of Beer benchmark is used to show whether the batches from Section A.2.1, with regards to `Datatypes`, and `Invocation` has the desired effect when generalizing to larger benchmarks.

The changes with regards to `Datatypes` is changing `int` variables to `uint` variables, as well as changing the method of `string` concatenation from `String.Format` to `StringBuilder`.

The changes with regards to `Invocation` is changing the `lambda` expressions to not use variables that are defined outside the `lambda` expressions and instead send the variables as parameters.

Code Changes

No changes to the code are necessary to create the default implementation of the flexible version of 99 Bottles of Beer within our framework.

The changes to datatypes are the same naive changes done in previous benchmarks. With regards to `Invocation` we change `lambda` expressions to use parameters instead of variables outside of the `lambda` expression.

```
1 public class Invocation {
2     public static int Invocation99BottlesOfBeer() {
3         for (ulong i = 0; i < LoopIterations; i++) {
4             ...
```

```
5               Func<string, string, string, string, string>
                ↪ describeBottles =
6                   (first, second, third, fourth) =>
                    ↪ string.Format("{0} {1}{2} of {3}", first,
                    ↪ second, third, fourth);
7               ...
8               for (int y = 0; y < 199; y++) {
9                   write(string.Format("{0} {1}, {0},",
                    ↪ describeBottles(describeCount(bottles),
                    ↪ Vessel, plural(bottles), Beverage),
                    ↪ Location));
10                  write(Act(ref bottles));
11                  write(string.Format("{0} {1}.",
                    ↪ describeBottles(describeCount(bottles),
                    ↪ Vessel, plural(bottles), Beverage),
                    ↪ Location));
12                  write(string.Empty);
13              }
14          }
15          return 2048;
16      }
17  }
```

Listing 16: Example of changing a `lambda` expression to using parameters.

In Listing 16 we can see how the `describeBottles` `lambda` expression has four parameters instead of one, making it possible to keep the `lambda` expression without references to outside variables.

An implementation with both these categories is also made, as in previous benchmarks.

Results

We look at the results of the different implementations of the 99 Bottles of Beer benchmark to see what effects the changes have on the energy consumption of this benchmark.

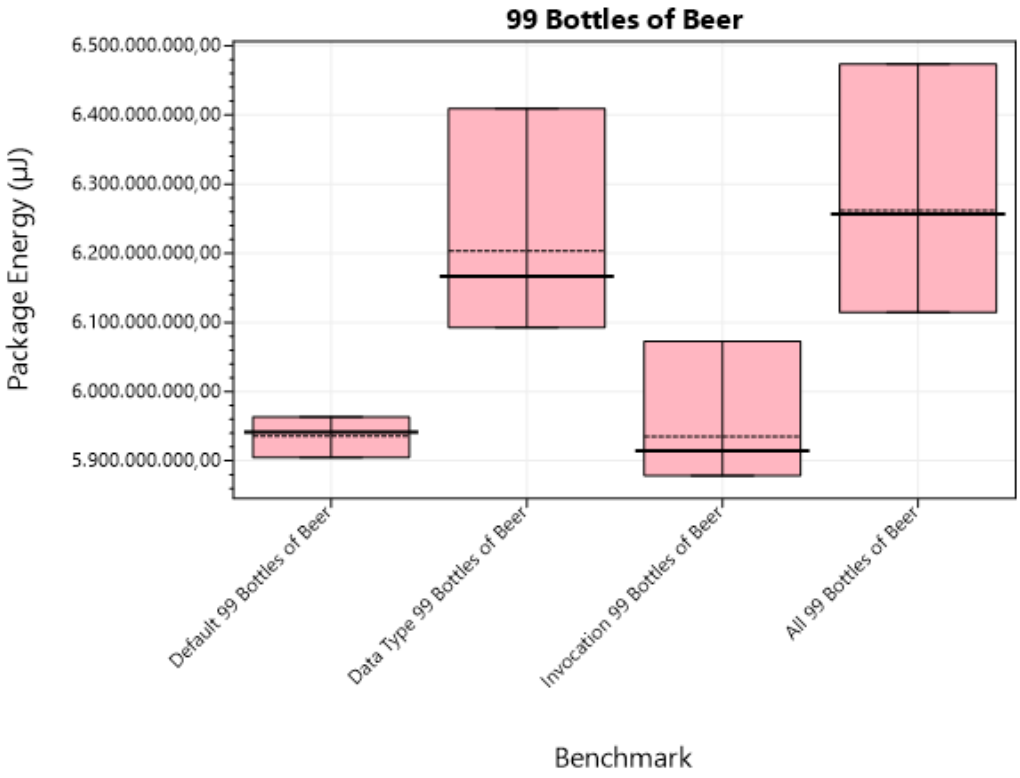


Figure 3.8: Energy consumption of each implementation of the 99 Bottles of Beer benchmark.

In Figure 3.8, we can see the energy consumption of the different implementations of the 99 Bottles of Beer benchmark. The y-axis does **not** start from zero and the span is around 650.000.000µJ. Here we can see that the `datatype` changes increases the energy consumption significantly from the default.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Default 99 Bottles of Beer	446.865	5.935.885.525	250.093.859	N/A
Data Type 99 Bottles of Beer	464.054	6.203.198.405	260.691.840	4,50%
Invocation 99 Bottles of Beer	445.618	5.935.232.282	249.346.889	-0,01%
All 99 Bottles of Beer	460.668	6.262.095.378	258.539.864	5,50%

Table 3.9: Table showing the elapsed time and energy measurement for each 99 Bottles of Beer, the difference from default is with regards to package energy.

In Table 3.9, we can see the same as in the plot, and also see that the changes to `invocation` is not significantly different from the default implementation.

To verify that the results are plausible, we do a sanity check on our package energy.

$$460.667{,}793470ms * 1000 * 20W = 9.213.355.869{,}4\mu J \tag{3.8}$$

In Equation 3.8 we can see that the sanity check on "All 99 Bottles of Beer" shows that there are no obvious errors to the result of the benchmark, as the result is within a factor of 2 to the actual result. The same is the case for the other benchmarks.

As the results show that the datatype changes have created an increase in energy consumption, we do an analysis of these results in Section 4.2.3.

3.2.6 Determine if a String has All the Same Characters

The benchmark [45] is an implementation of an algorithm that checks if a string consists of only the same characters or if there are differences in the string.

The "Determine if a String has All the Same Characters" benchmark is used to show whether the batches from Section A.2.1, with regards to `Datatypes`, and `Loops` has the desired effect when generalizing to larger benchmarks.

The change with regards to `Datatypes` is changing the method of `string` concatenation when using `Console.WriteLine`, to using `StringBuilder`.

The change with regards to `Loops` is changing the `for` loop to a `foreach`.

Code Changes

No changes to the code are necessary to create the default implementation of the "Determine if a String has All the Same Characters" benchmark.

The changes to the code in the `Datatypes` batch and the `Loops` batch are the same naive changes done in previous benchmark, where methods of `string` concatenation is changed to `StringBuilder`, and the `for` loops is changed to `foreach` loops when iterating through the string.

Furthermore, a benchmark with all the changes is created like in previous benchmarks.

Results

We compare the results of the different implementations of the "Determine if a String has All the Same Characters" benchmark, to get further insight

into how these changes affect larger benchmarks.

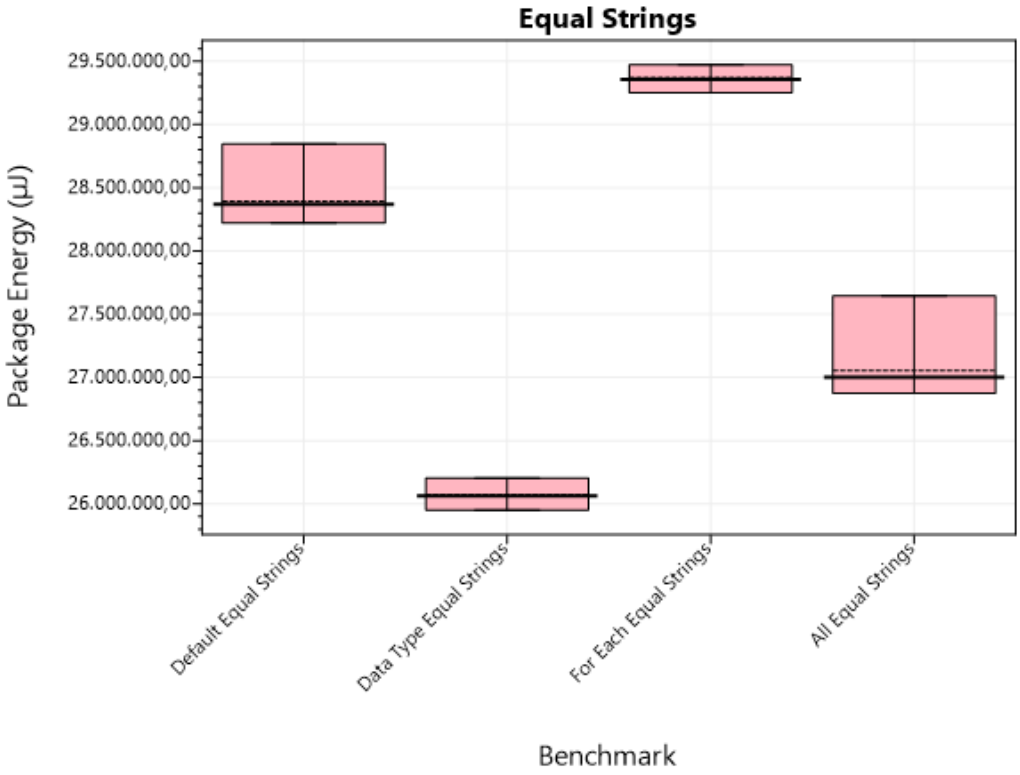


Figure 3.9: Energy consumption of each implementation of the "Determine if a String has All the Same Characters" benchmark.

In Figure 3.9, we can see the energy consumption of the different implementations of the benchmark. The y-axis does **not** start from zero and the span is around 4.000.000µJ. Here we can see that changes to `datatypes` have a significant decrease in energy consumption, while the changes to `Loops` have a significant increase in energy consumption, which is surprising.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Default Equal Strings	1.997	28.393.107	1.112.824	N/A
Data Type Equal Strings	1.827	26.071.351	1.037.025	-8,18%
For Each Equal Strings	2.059	29.370.734	1.149.146	3,44%
All Equal Strings	1.903	27.055.465	1.078.427	-4,71%

Table 3.10: Table showing the elapsed time and energy measurement for each Equal Strings, the difference from default is with regards to package energy.

In Table 3.10 we can see the same as in the plot, that there are more than

2% difference between the default and the other variants with regards to energy consumption.

To verify that the results are plausible, we create a sanity check on our package energy.

$$2.059{,}078895ms * 1000 * 20W = 41.181.577{,}9\mu J \tag{3.9}$$

In Equation 3.9 we can see that the sanity check shows that there are no obvious errors to the result of the `foreach` benchmark, as the result is within a factor of 2 to the actual result. The same is the case for the other benchmarks.

As the results show that the `loops` changes have created an increase in energy consumption, we do an analysis of these results in Section 4.2.4.

3.2.7 Dijkstra's Algorithm

The benchmark [46] is an implementation of Dijkstra's Algorithm.

We use this benchmark to show whether the batches from Section A.2.1, with regards to `Datatypes`, LINQ, `Collections`, and `Objects` has the desired effect when generalizing to larger benchmarks.

The change with regards to `Datatypes` is changing `int` variables to `uint` variables.

The change with regards to LINQ is removing and replacing cases of LINQ where possible.

The change with regards to `Collections` is changing `Lists` to `arrays` where possible.

The change with regards to `Objects` is changing `class` to `struct` where possible.

Code Changes

No changes to the code are necessary to create the default implementation of Dijkstra's Algorithm.

The changes with regards to `datatypes`, `LINQ`, and `objects`, are the same naive changes as done in previous benchmarks. With regards to `Collections` we change occurrences of `List` to `array` when possible.

```
1 public class Collections {
2     internal sealed class Graph {
```

```
3           private readonly EdgeList[] adjacency;
4           public Graph(int vertexCount) {
5               adjacency = new EdgeList[vertexCount];
6               for (var index = 0; index < adjacency.Length;
                ↪  index++) {
7                   adjacency[index] = new EdgeList();
8               }
9           }
10          ...
11      }
12  }
```

Listing 17: Example of changing a `List` to an `array`, `Graph` class.

In Listing 17 we can see how the `adjancency List` has been changed into an `array`, and how the corresponding constructor for `Graph` has been changed to accomodate this change.

An implementation with all these categories is also made, as in previous benchmarks.

Results

We look at the results of the different implementations of Dijkstra's Algorithm to see what effects our changes have on the energy consumption of this benchmark.

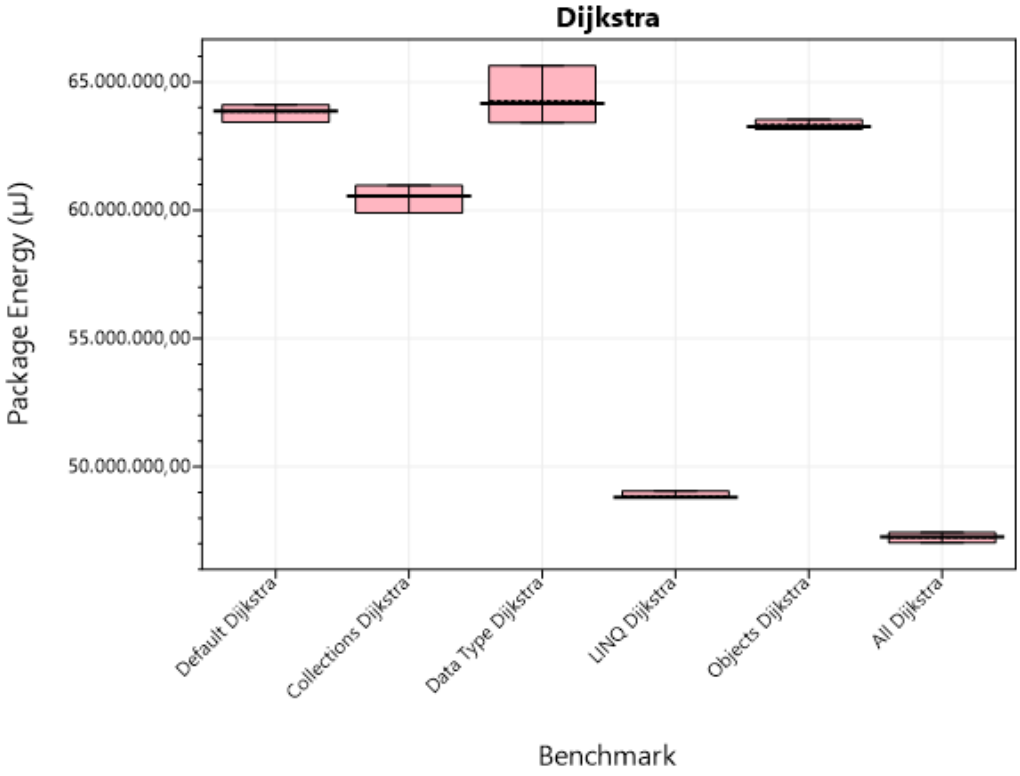


Figure 3.10: Energy consumption of each implementation of Dijkstra's Algorithm.

In Figure 3.10, we can see the energy consumption of the different implementations of Dijkstra's Algorithm benchmark. The y-axis does **not** start from zero and the span is around 20.000.000µJ. Here we can see that the `collections` and `LINQ` changes have a significant effect on the energy efficiency of the benchmark, while the other changes have less effect.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Default Dijkstra	4.469	63.836.619	2.511.046	N/A
Collections Dijkstra	4.258	60.555.783	2.396.288	-5,14%
Data Type Dijkstra	4.508	64.258.298	2.538.355	0,66%
LINQ Dijkstra	3.423	48.861.610	1.917.692	-23,46%
Objects Dijkstra	4.446	63.341.985	2.496.351	-0,77%
All Dijkstra	3.332	47.226.367	1.868.384	-26,02%

Table 3.11: Table showing the elapsed time and energy measurement for each Dijkstra, the difference from default is with regards to package energy.

In Table 3.11, we see the same as in the plot, where `collections` and `LINQ` have a significant impact on the efficiency, while the other changes

have shown no significant difference.

To verify that the results are plausible, we do a sanity check on our package energy.

$$4.468{,}567573ms * 1000 * 20W = 89.371.351{,}46\mu J \tag{3.10}$$

In Equation 3.10 we can see that the sanity check shows that there are no obvious errors to the result of the default benchmark, as the result is within a factor of 2 to the actual result. The same is the case for the other benchmarks.

As the results show the same as earlier results with regards to `objects`, and otherwise shows what is expected, we do not analyze these results.

3.2.8 Happy Numbers

The benchmark [47] is an implementation of a solution to find and print the first 8 happy numbers. A happy number is defined by a positive integer which is replaced by the squares of its digits, repeating until the number equals 1. If it does not reach 1 it will cycle endlessly and is an unhappy number.

The Happy Number benchmark is chosen to show whether the batches from Section A.2.1 with regards to `Datatypes` and `Collections` can be generalized to a larger benchmark.

The change within `Datatypes` is changing the `int` variables to `uint` and changing the strategy of `string` concatenation from using the concatenation operator (+) to using `StringBuilder`.

The change within `Collections` is changing `List` to `Array`.

Code Changes

No changes to the code are necessary to create the default implementation of the Happy Numbers benchmark.

The changes with regards to `datatype`, and `collections` are the same naive changes as done in previous benchmarks.

An implementation where both of these batches of changes are done is also created like in previous benchmarks.

Results

We look at the different implementations of the Happy Numbers benchmark to see if our changes affect the energy consumption of the benchmark.

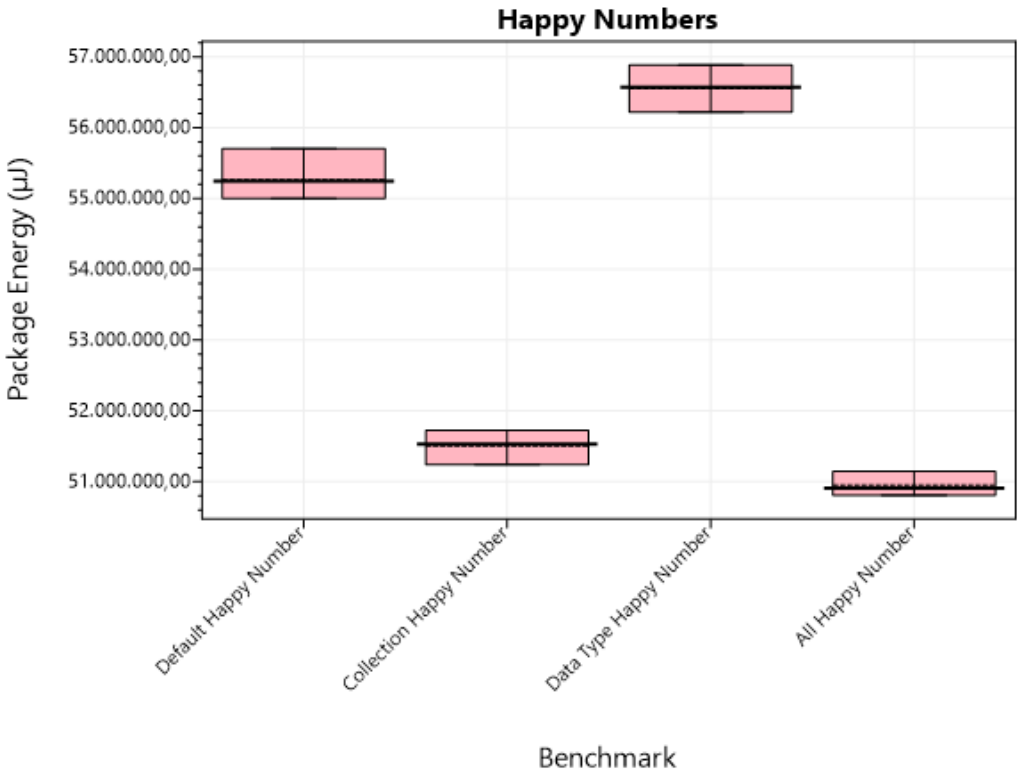


Figure 3.11: Energy consumption of each implementation of the Happy Numbers benchmark.

In Figure 3.11, we can see the energy consumption of the different implementations of the Happy Numbers benchmark. The y-axis does **not** start from zero to show a better view of the differences, and the span is around 6.000.000µJ. Here we can see that the `collections` changes decrease the energy efficiency of the benchmark significantly, while the `datatype` changes significantly increases the energy efficiency of the benchmark.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Default Happy Number	4.171	55.261.641	2.349.720	N/A
Collection Happy Number	3.903	51.505.524	2.186.327	-6,80%
Data Type Happy Number	4.278	56.554.523	2.406.531	2,34%
All Happy Number	3.872	50.953.873	2.171.529	-7,80%

Table 3.12: Table showing the elapsed time and energy measurement for each Happy Numbers, the difference from default is with regards to package energy.

In Table 3.12, we can see that the same as in the plot, where the energy efficiency changes are all above 2% from the default implementation. Interesting to note is that the change in efficiency when using all the changes are more significant than only having the `collection` changes, even though the `datatype` changes seems to create an increase in energy consumption. That being said, the difference between all changes and `collection` changes are not significant, as they are within 2% of each other.

To verify that the results are plausible, we do a sanity check on our package energy.

$$3.903{,}189256ms * 1000 * 20W = 78.063.785{,}12\mu J \tag{3.11}$$

In Equation 3.11 we can see that the sanity check shows that there are no obvious errors to the result of the `collection` benchmark, as the result is within a factor of 2 to the actual result. The same is the case for the other benchmarks.

As the results show that there are different impacts when using all the changes, we look into this in Section 4.2.5.

3.2.9 Introspection

Introspection [48] is a task that is used to verify that a compiler is up-to-date and check whether a variable exists and the method `abs` exists on it, the specifics of which can be seen in [48].

The Introspection benchmark is chosen to show whether the batches from Section A.2.1 with regards to `Datatypes`, and `Invocation` have the desired results when generalized to a larger benchmark.

The changes to the benchmark with regards to `Datatypes` is to replace `int` variables with `uint` variables. Furthermore, changes with regards to `Console.WriteLine string` concatenation are made, specifically changing this from `String.Format` to `StringBuilder`.

The changes to the benchmark with regards to `Invocation` is to replace `Reflection` with `Reflection Delegation`.

Code Changes

To make it possible to run the benchmark as part of our framework, it is necessary to make changes besides the ones needed to test the language constructs.

Specifically, we only look through the class that is part of the specific benchmark, instead of all the exported types.

```
1 public class Default {
2     public static void Main() {
3         ...
4         foreach (FieldInfo field in typeof(Default).GetFields())
          ↪ {
5             ...
6         }
7         ...
8     }
9 }
```

Listing 18: Change to Default implementation of Introspection, looking through `Default` class.

In Listing 18 on line 4 we can see that only the `Default` class is looked through to get the fields necessary for the benchmark. This is also done at the end of the `Main` method.

Besides these changes, it has been necessary to add an extra check for the assemblies, so that we only get the assemblies that are not dynamic.

```
1 public class Default {
2     public static void Main() {
3         ...
4         foreach (Assembly refAsm in
          ↪ AppDomain.CurrentDomain.GetAssemblies().Where(assembly
          ↪ => !assembly.IsDynamic)) {
5             ...
6         }
7         ...
```

```
8       }
9   }
```

Listing 19: Change to getting assemblies, so only non-dynamic assemblies are collected.

In Listing 19 on line 4 we can see that the `.Where(assembly => !assembly.IsDynamic)` LINQ method has been appended to the line, to only get the non-dynamic assemblies.

Besides these changes, the changes with regards to `datatype` are the same naive changes as before, while the `invocation` change wraps a reflection invocation in a `delegate`.

```
1   public class Invocation {
2       public static void Main() {
3           ...
4           foreach (Assembly refAsm in
            ↪  AppDomain.CurrentDomain.GetAssemblies().Where(assembly
            ↪  => !assembly.IsDynamic)) {
5               foreach (Type type in refAsm.GetExportedTypes()) {
6                   if (type.Name == "Math") {
7                       MethodInfo? absMethod =
                        ↪  type.GetMethod("Abs", new Type[] {
                        ↪  typeof(int) });
8                       if (absMethod != null) {
9                           var absDelegate = (Func<int,
                            ↪  int>)Delegate.CreateDelegate(typeof(Func<int,
                            ↪  int>), absMethod);
10                          Console.WriteLine("bloop's abs value =
                            ↪  {0}",
                            ↪  absDelegate((int)bloopField.GetValue(null)));
11                      }
12                  }
13              }
14          }
15          ...
16      }
17  }
```

Listing 20: Example of changing a reflection invocation to a `delegate`.

In Listing 20, we can see how the `MethodInfo` gathered via reflection is wrapped in a `delegate` on line 9, and is called on line 10.

An implementation where both the `datatype` changes and the `invocation` changes are used, is created like in previous benchmarks.

Results

We look at the different implementations of the `Introspection` benchmark to see if the changes affect the energy consumption of the benchmark.

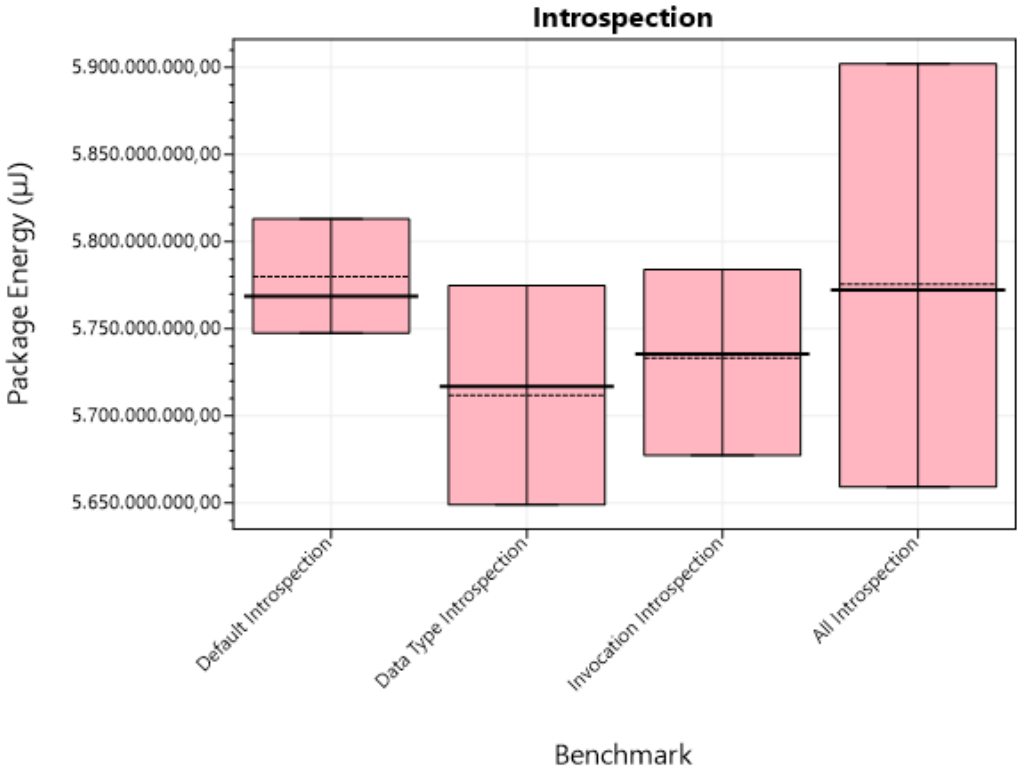


Figure 3.12: Energy consumption of each implementation of the `Introspection` benchmark.

In Figure 3.12, we can see the energy consumption of the different implementations of the `Introspection` benchmark. The y-axis does **not** start from zero to show a better view of the differences, and the span is around 400.000.000µJ. Here we can see that the changes have no significant impact on the energy consumption of the benchmark.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Default Introspection	442.800	5.779.993.273	248.941.406	N/A
Data Type Introspection	443.159	5.711.772.678	249.279.297	-1,18%
Invocation Introspection	442.916	5.733.289.931	248.895.182	-0,81%
All Introspection	444.107	5.775.570.984	250.320.251	-0,08%

Table 3.13: Table showing the elapsed time and energy measurement for each Introspection, the difference from default is with regards to package energy.

In Table 3.13, we can see the same as in the plot, all of the results are within 2% of the default implementation, meaning no significant difference is observed with the changes as they currently stand.

To verify that the results are plausible, we do a sanity check on our package energy.

$$444.106{,}762695ms * 1000 * 20W = 8.882.135.253{,}9\mu J \tag{3.12}$$

In Equation 3.12 we can see that the sanity check shows that there are no obvious errors in the result of the "All Introspection" benchmark, as the result is within a factor of 2 to the actual result. The same is the case for the other benchmarks.

As the results show that there is no impact of the change to `invocation`, despite a large impact in the relevant microbenchmark, we look into this in Section 4.2.6.

3.2.10 World Cup Group Stage

This benchmark [49] is a solution to the group stage of the football world cup, where four teams in a group should play every other team in the group once. The results for these games determine which teams advance to the "knockout stage". The two teams from each group with the most points advance.

The World Cup Group Stage benchmark is chosen as it has possible changes with regards to `Datatypes` and several LINQ implementations, which are not optimized for energy consumption according to the results of our microbenchmarks.

The changes to the benchmark with regards to `Datatype` is replacing `int` variables with `uint` variables, and changing `String.Concat` and `string` interpolation to `StringBuilder`.

The changes to the benchmark with regards to LINQ is replacing it with any other functionality possible.

Code Changes

The only change to the default implementation of this benchmark is that a `Console.Read` at the end of the program has been removed, as this would otherwise halt the benchmark during measurement.

The same naive changes to `datatypes`, and `LINQ` are done, as in previous benchmarks.

Furthermore, a benchmark where both of these batches are done is also created to see the impact of this, like in previous benchmarks.

Results

We look at the different implementations of the World Cup Group Stage benchmark to see if the changes have an impact on energy consumption, like in previous benchmarks.

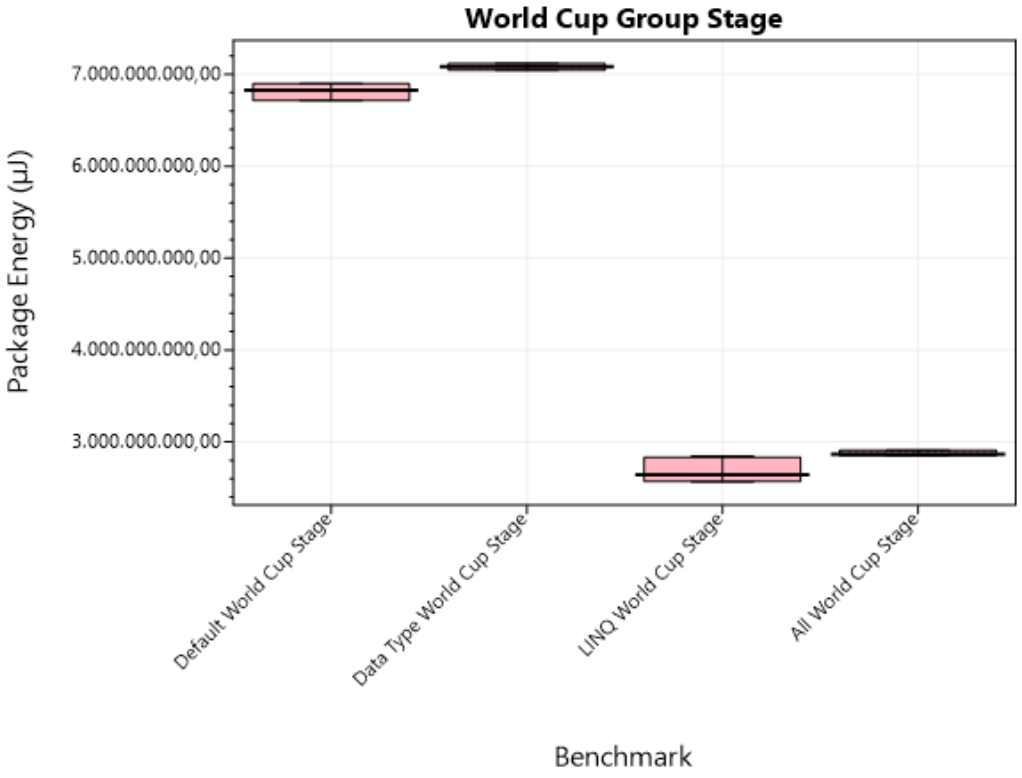


Figure 3.13: Energy consumption of each implementation of the World Cup Group Stage benchmark.

In Figure 3.13, we can see the energy consumption of the different implementations of the World Cup Group Stage benchmark. The y-axis does **not** start from zero to show a better view of the differences, and the span is around 4.500.000.000µJ. Here we can see that the `LINQ` benchmark has a significant decrease in energy consumption compared to the default implementation, while the `datatype` benchmark has a significant increase in energy consumption compared to the default implementation.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Default World Cup Stage	480.730	6.815.760.525	271.617.730	N/A
Data Type World Cup Stage	502.122	7.082.867.839	283.386.285	3,92%
LINQ World Cup Stage	199.663	2.649.373.573	112.157.494	-61,13%
All World Cup Stage	215.450	2.873.367.025	121.089.844	-57,84%

Table 3.14: Table showing the elapsed time and energy measurement for each World Cup Group Stage, the difference from default is with regards to package energy.

In Table 3.14 we can see the same as in the plot. Furthermore, it is confirmed that the `datatype` benchmark increases the energy consumption significantly, as the difference to the default implementation is above 2%.

To verify that the results are plausible, we do a sanity check on the package energy of the benchmarks.

$$199.663{,}256073ms * 1000 * 20W = 3.993.265.121{,}46\mu J \tag{3.13}$$

In Equation 3.13 we can see that the sanity check shows that there are no obvious errors to the result of the `LINQ` benchmark, as the result is within a factor of 2 to the actual result. The same is the case for the other benchmarks.

As the results show that there is a significant increase in energy consumption when using the `datatype` changes, we look into this in Section 4.2.7.

3.2.11 Summary

In this section we summarize the changes and results of each large benchmark, to serve as an overview for the analysis. These results are used as a starting point for the analysis, which makes it possible to create formal suggestions for *Energy Analyzer* that helps developers write more energy efficient code.

Benchmark	Variants	Results Package E (µJ)	Difference from Default
2048	Default	8.171.909.912	N/A
	All	8.186.948.567	0,18%
	DataType	8.216.815.681	0,55%
	Foreach	8.178.229.403	0,08%
	Object	8.104.721.752	-0,82%
	Switch	8.064.600.838	-1,31%
21 Game	Default	58.685.175	N/A
	All	67.220.655	14,54%
	DataType	68.117.482	16,07%
	Switch	58.570.673	-0,20%
	Exception	58.369.766	-0,54%
Four Squares	Default	23.477.282.118	N/A
	All	14.495.177.083	-38,26%
	DataType	23.264.401.475	-0,91%
	LINQ	14.662.122.395	-37,55%
99 Bottles of Beer	Default	5.935.885.525	N/A
	All	6.262.095.377	5,50%
	DataType	6.203.198.404	4,50%
	Invocation	5.935.232.282	-0,01%
Determine if a String has All the Same Characters	Default	28.393.107	N/A
	All	27.055.465	-4,71%
	DataType	26.071.351	-8,18%
	Foreach	29.370.734	3,44%
Dijkstra's Algorithm	Default	63.836.619	N/A
	All	47.226.366	-26,02%
	DataType	64.258.298	0,66%
	Object	63.341.985	-0,77%
	LINQ	48.861.610	-23,46%
	Collections	60.555.782	-5,14%
Happy Number	Default	55.261.640	N/A

	All	50.953.872	-7,80%
	DataType	56.554.523	2,34%
	Collections	51.505.523	-6,80%
Introspection	Default	5.779.993.272	N/A
	All	5.775.570.983	-0,08%
	DataType	5.711.772.677	-1,18%
	Invocation	5.733.289.930	-0,81%
World Cup Group Stage	Default	6.815.760.525	N/A
	All	2.873.367.024	-57,84%
	DataType	7.082.867.838	3,92%
	LINQ	2.649.373.573	-61,13%

Table 3.15: Summary of all larger benchmarks

Chapter 4

Benchmark Analysis

In this chapter we look into and analyse the results of the benchmark, and if any changes needs to be made to gather better results. The result of this analysis is formal suggestions that are used in *Energy Analyzer* to help developers write more energy efficient code.

4.1 Microbenchmark Analysis

In this section we analyze the results for microbenchmarks gathered in Chapter 3, we use the same overall process as used in [3, p.111], with a few differences.

This provides insight into why the results occurred in the way they did and thereby makes it possible to create more precise suggestions for *Energy Analyzer* that will help developers write energy efficient code.

We add an extra step in the process presented in [3, p.111], specifically before checking the Intermediate Language (IL) code, we check if the construct we are testing is a library construct, in which case we study how it is implemented in C#.

Besides this, we discovered the tool PowerUp [50] that can be used for checking the assembly code in release mode instead of debug mode, therefore we change the step regarding checking the assembly code, which means that the overall process now is the following:

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 removed by the compiler due to optimizations.

2. Read the documentation for the language constructs we are looking at to see if there is an explanation for differing energy consumption.
3. If the construct is a library construct, check the implementation in C#.
4. Look at the IL code to see if there is a difference between the benchmarks.
5. Look at the assembly code in release mode to see if there is a difference between the benchmarks.

4.1.1 Lambda Expressions Outside Loop Analysis

In this section, we look at why there is a variance in the results for `lambda expressions` that are created outside the `for` loop.

As mentioned in Section 3.1.1, the results from the lambda expressions outside the loop were inconsistent between runs, so the procedure for these results is different from the ordinary analysis procedure.

First, we establish that the results are inconsistent between runs by running these specific benchmarks 10 times.

Package Energy (μJ)	Lambda Parameter	Lambda	Lambda Action	Lambda Delegate	Lambda Closure
Run 1	20.954	24.740	22.353	18.379	22.196
Run 2	18.245	18.319	19.581	18.209	21.905
Run 3	18.476	24.247	19.771	21.062	19.490
Run 4	20.898	18.338	21.627	18.187	19.256
Run 5	20.947	18.122	21.551	20.967	21.559
Run 6	20.903	17.990	19.507	17.974	19.216
Run 7	20.876	23.445	21.345	20.825	19.077
Run 8	20.607	17.891	20.951	20.661	21.083
Run 9	20.980	23.933	21.876	18.070	21.357
Run 10	20.849	18.056	21.437	20.943	21.269

Table 4.1: The results of running the lambda expressions outside loop microbenchmarks 10 times.

In Table 4.1 we can see all of the benchmarks have a large variance.

Percentage Variance	
Lambda Parameter	13,04%
Lambda	27,68%
Lambda Action	12,73%
Lambda Delegate	14,66%
Lambda Closure	14,05%

Table 4.2: The percentage variance between the highest and lowest energy consumption measured for each of the benchmarks.

In Table 4.2 we can see the variance between the highest and lowest energy consumption measured for each benchmark.

The variance is higher than the 2% ordinary results shown in [3], we hypothesize that there is another reason for this variance is occurring, therefore we do not start by looking at the differences between the individual benchmarks.

As `Func`, and `Action` utilizes `delegate` [51, 52, 53], and as `delegate` shows the same variance as the rest, we will focus on the `Lambda Delegate` benchmark.

The cause of the variance can be within two categories, either `delegate` is inconsistent between each measurement, or `lambda expressions` is inconsistent between each measurement. This is the case as the only other parts to the `LambdaDelegate` benchmark, are addition and the `for` loop, which we know from [3] does not cost this much to do. Because of this, we create a new benchmark where we use a `delegate` with a named function instead of a `lambda expression`. If the new benchmark is also inconsistent, we assume that `delegate` is what causes the issues with the results.

```
1  public class LambdaBenchmarks {
2      ...
3      private delegate ulong PerformCalculation();
4      [Benchmark("Lambda Expression", "Delegate Test")]
5      public static ulong DelegateTest() {
6          ulong result = 0;
7          PerformCalculation test = Test;
8          for (ulong i = 0; i < LoopIterations; i++) {
9              result = test() + result + i;
10         }
11         return result;
```

```
12      }
13      public static ulong Test() {
14          return 25;
15      }
16      ...
17  }
```

Listing 21: The `DelegateTest` method which tests a `delegate` with a named benchmark in the `LambdaBenchmarks` class.

We run this benchmark 10 times to get an idea of the variance with `delegate`, so we can ascertain whether `delegate` or `lambda expressions` are the cause of the variance.

Package Energy (μJ)	Delegate Test
Run 1	23.988
Run 2	23.958
Run 3	26.539
Run 4	24.081
Run 5	24.243
Run 6	26.895
Run 7	24.039
Run 8	26.728
Run 9	26.575
Run 10	26.591

Table 4.3: The results of running the delegate test benchmark 10 times.

In Table 4.3 we can see the results of running the `DelegateTest` benchmark 10 times, which shows that `delegate` is the cause of the variance, where there is a difference of 10,92% between the lowest and highest energy consumption.

Because of this we focus on the documentation for `delegate` [53] to try to find an explanation.

The documentation states "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 could hint towards the issue being with garbage collection of the object instances.

To look into this, we create a critical area around the runs of the benchmarks where garbage collection will not be done (if possible), by using `GC.TryStartNoGCRegion` [54] and `GC.EndNoGCRegion` [55].

As the amount of memory used will increase as each benchmark is being run, we limit our scope to looking at the `Lambda` benchmark, to ensure that the "No Garbage Collection Region" can be utilized successfully. This means, we skip all the other benchmarks, and add the lines of code around the relevant parts of the code that runs the benchmark.

Package Energy (μJ)	Lambda Benchmark
Run 1	18.093
Run 2	17.933
Run 3	18.127
Run 4	18.093
Run 5	17.985
Run 6	18.047
Run 7	17.984
Run 8	18.003
Run 9	18.051
Run 10	18.008

Table 4.4: The results of running the lambda benchmark 10 times with garbage collection turned off.

As can be seen in Table 4.4, this has made the results consistent, making the variance drop from 27,68% to 1,07%. This does not guarantee consistent results in all cases, as garbage collection is not completely turned off, the runtime only tries to not garbage collect during the critical region. If the critical region exceeds a specific amount of memory, the garbage collection will occur anyway and an outlier will happen. Furthermore, there are a lot of other things that influence the energy consumption when using a lot of memory, therefore outliers may still occur.

This is an explanation for why the variance is so large when running the benchmarks ordinarily. Getting results without the effects of memory management is not possible for all of the benchmarks within this group, as some of them use more memory than others. Because of this, either garbage collection will trigger or other forms of memory management will trigger making the results vary significantly.

4.1.2 Lambda Expressions Inside Loop Analysis

In this section, we look into why there is a difference in energy consumption for different types of `lambda expressions` that are created within the `for` loop of the benchmarks.

Specifically, we want to look into why:

- Using the `Action` construct and using closure with a lambda expression is less efficient than not doing that when initializing the lambda expressions inside the `for` loop.

First we look at how the benchmarks are written, with one of them shown in Section 3.1.1, and the rest shown in [1] we can see that the `Action` and `Closure` benchmarks that a variable initialized outside the `for` loop is utilized inside the `lambda expression` in the `for` loop, which is the main difference between the benchmarks with a high energy consumption and a low energy consumption.

This could be the reason why there is a difference, so we look at [56], specifically at the section called "Capture of outer variables and variable scope in lambda expressions". The documentation states "Variables that are captured in this manner are stored for use in the lambda expression even if the variables would otherwise go out of scope and be garbage collected.", which makes sense as an explanation for why these would increase the energy consumption. It makes sense that storing variables every time a new `lambda expression` is created costs more energy, compared to not having to do that.

We consider this an explanation for why there is a difference in energy consumption between `InsideLoopLambdaAction` and `InsideLoopLambdaClosure` compared to `InsideLoopLambda`, `InsideLoopLambdaDelegate`, and `InsideLoopLambdaParameter`.

4.1.3 Exception Creation Analysis

In this section, we look into why there is a difference in energy consumption for creating different exceptions. We specifically want to look into why:

- Creating an `Exception` consumes 8 times less package energy than creating an `ArgumentException` or a `DivideByZeroException`.

First, we look at how the benchmarks are written which are shown in Section 3.1.2. We can see the benchmarks are written in the same way, except for the type of `exception`. Therefore the reason for the difference in

energy consumption does not come from the benchmarks written in different ways, and we look into the documentation for an explanation.

In Microsoft's Documentation [57] `Exception` is the base for all exceptions. In [58] we see `ArgumentException` inherits from `SystemException` class, which inherits from the `Exception` class. While in [59] we see `DivideByZeroException` inherits from `ArithmeticException` which inherits from `SystemException`. `ArgumentException` and `DivideByZeroException` inherits from `Exception` class, this could be a cause for higher energy consumption. Besides this, there could be more functionality which could be an additional cause for higher energy consumption.

To get further insight we look into the library implementation of the exceptions. Looking into the different implementations of the exceptions, we find nothing that was not explained in the documentation. Therefore we look into the IL code generated from the benchmarks.

```
1 ...
2 // result = new ArgumentException();
3 IL_000b: newobj       instance void
  ↪  [System.Private.CoreLib]System.ArgumentException::.ctor()
4 IL_0010: stloc.0
5 ...
```

Listing 22: The `CreateArgumentException` method translated to IL code.

In Listing 22 we see the IL code for `CreateArgumentException`. Here we see what type of `exception` is created.

```
1 ...
2 // result = new DivideByZeroException();
3 IL_000b: newobj       instance void
  ↪  [System.Private.CoreLib]System.DivideByZeroException::.ctor()
4 IL_0010: stloc.0
5 ...
```

Listing 23: The `CreateDivideByZeroException` method translated to IL code.

In Listing 23 we see the IL code for `CreateDivideByZeroException`. As the case with the C# code, it is very similar to the IL code of `CreateArgumentException`, except that we create a `DivideByZeroException`.

```
...
// result = new Exception();
IL_000b: newobj       instance void
  ↪  [System.Private.CoreLib]System.Exception::.ctor()
IL_0010: stloc.0
...
```

Listing 24: The `CreateException` method translated to IL code.

In Listing 24 we see the IL code for `CreateException`. Again the only difference from the other two code snippets is that the benchmark creates an `Exception`. The IL code shows the same as the C# code, the only difference between the benchmarks is which type of exception is used, therefore we look into the assembly code.

```
ExceptionBenchmark+Exception CreateException():
...
7FFC05A90D26: mov    rcx, 7FFC0360F048h
7FFC05A90D30: call   CORINFO_HELP_NEWSFAST 7FFC6310AE20
7FFC05A90D35: mov    rsi, rax
7FFC05A90D38: mov    dword ptr [rsi+70h], 0E0434352h
7FFC05A90D3F: mov    dword ptr [rsi+74h], 80131500h
...
```

Listing 25: The `CreateException` method translated to ASM code. Instruction Count: 30; Code Size: 121.

In Listing 25 we see the relevant parts of the assembly code for `CreateException`, this is the benchmark that consumes the least amount of energy. The benchmark consists of 30 code instructions, with the relevant part within the loop being 5 code instructions.

```
ExceptionBenchmark+Exception CreateDivideByZeroException():
...
7FFC05A90C67: mov    rcx, 7FFC05A8D5C0h
7FFC05A90C71: call   CORINFO_HELP_NEWSFAST 7FFC6310AE20
7FFC05A90C76: mov    rsi, rax
```

```
6  7FFC05A90C79: call   System.SR.get_Arg_DivideByZero()
   ↪  7FFC03DC7AF0
7  7FFC05A90C7E: mov    dword ptr [rsi+70h], 0E0434352h
8  7FFC05A90C85: mov    dword ptr [rsi+74h], 80131500h
9  7FFC05A90C8C: lea    rcx, [rsi+10h]
10 7FFC05A90C90: mov    rdx, rax
11 7FFC05A90C93: call   CORINFO_HELP_ASSIGN_REF 7FFC6310AA00
12 7FFC05A90C98: mov    dword ptr [rsi+74h], 80131501h
13 7FFC05A90C9F: mov    dword ptr [rsi+74h], 80070216h
14 7FFC05A90CA6: mov    dword ptr [rsi+74h], 80020012h
15 ...
```

Listing 26: The `CreateDivideByZeroException` method translated to ASM code. Instruction Count: 37; Code Size: 161.

In Listing 26 we see the assembly code for `CreateDivideByZeroException`. The benchmark has 37 instructions, which is 7 more than `CreateException`, and the relevant part within the loop is 12 instructions. We expect this to be the reason `CreateDivideByZeroException` consumes more energy, as extra instructions take more time and consume more energy. The instruction on line 6 and 11 are calls that are not present in the `CreateException` benchmark, which is a reason for the high energy consumption, as calling a method means code not seen here is executed and therefore consumes more energy.

```
1  ExceptionBenchmark+Exception CreateArgumentException():
2  ...
3  7FFC05A90BA7: mov    rcx, 7FFC03614B40h
4  7FFC05A90BB1: call   CORINFO_HELP_NEWSFAST 7FFC6310AE20
5  7FFC05A90BB6: mov    rsi, rax
6  7FFC05A90BB9: call   System.SR.get_Arg_ArgumentException()
   ↪  7FFC03DC7A20
7  7FFC05A90BBE: mov    dword ptr [rsi+70h], 0E0434352h
8  7FFC05A90BC5: mov    dword ptr [rsi+74h], 80131500h
9  7FFC05A90BCC: lea    rcx, [rsi+10h]
10 7FFC05A90BD0: mov    rdx, rax
11 7FFC05A90BD3: call   CORINFO_HELP_ASSIGN_REF 7FFC6310AA00
12 7FFC05A90BD8: mov    dword ptr [rsi+74h], 80131501h
```

```
13 7FFC05A90BDF: mov    dword ptr [rsi+74h], 80070057h
14 ...
```

Listing 27: The `CreateArgumentException` method translated to ASM code. Instruction Count: 36; Code Size: 154.

In Listing 27 we see the assembly code for `CreateArgumentException`. The benchmark has 36 instructions, 6 more than `CreateException`, and the relevant part within the loop being 11 instructions. We expect this is the reason `CreateArgumentException` consumes more energy than `CreateException`. Again, the instructions on line 6 and 11 are call instructions not present in the `CreateException` benchmark, which is a reason for the extra energy consumption, as mentioned before. We consider the extra instructions and the fact that `ArgumentException` and `DivideByZeroException` inherit from `Exception`, to be part of the explanation for the difference between creating an `Exception` compared to creating an `ArgumentException` or a `DivideByZeroException`.

4.1.4 Throwing and Catching Exceptions Analysis

In this section, we look into why there is a difference in energy consumption for throwing and catching different exceptions. We specifically want to look into why:

- Throwing and catching an `Exception` is more energy efficient than using an `ArgumentException` or a `DivideByZeroException` when not using cached exceptions.
- Throwing and catching a cached exception consumes less energy than throwing and catching a newly created exception when using `ArgumentException` or `DivideByZeroException`.

We start by looking into why throwing and catching an `Exception` is more energy efficient than using an `ArgumentException` or a `DivideByZeroException` when not using cached exceptions. We already have an answer to this from Section 4.1.3, where we found that creating an `Exception` consumes less energy than creating an `ArgumentException` or a `DivideByZeroException`, because creating `ArgumentException` and `DivideByZeroException` have more instructions.

We now look into why throwing and catching a cached exception consumes less energy than throwing and catching a newly created exception when using `ArgumentException` or `DivideByZeroException`. First, we look into how the benchmarks are written which are shown in Section 3.1.2. We can see that the benchmarks consuming the least amount of energy are the ones using cached exceptions. Using cached exceptions means we do not need to initialize a new exception every time we want to throw an exception, and therefore we save energy. We consider this a satisfactory explanation as to why using a cached exception is cheaper than using a new exception.

4.2 Larger benchmark Analysis

In this section, we analyze the results for the larger benchmarks gathered in Section 3.2. This is done to get more insight into the results, and thereby make it possible to give better suggestions for *Energy Analyzer* that will help developers write more energy efficient code.

We use the following process:

1. Look for the cause of the oddities by making various changes to the relevant benchmarks.
2. Try to create an optimal benchmark with the knowledge gathered. An optimal benchmark is a benchmark written using all of the improved suggestions we have found from the analysis of the larger benchmarks.
3. Try to create a suggestion general enough for *Energy Analyzer*, while not creating an increase in energy consumption.

4.2.1 2048 Analysis

In Section 3.2.2, we found that the changes made to `2048` had no significant effect on the energy consumption of the program. To figure out why this is, we look into what changes were made and what consequences these had to make the program work as intended.

When looking at the datatype changes, we see that changing `int` variables to `uint` variables, there are cases where casts to `uint` is necessary.

```
public class DataType {
    internal class G2048 {
        private void AddTile() {
            for (uint y = 0; y < 4; y++) {
                for (uint x = 0; x < 4; x++) {
                    if (_board[x, y].Value != 0) continue;
                    uint a, b;
                    do {
                        a = (uint)_rand.Next(0, 4);
                        b = (uint)_rand.Next(0, 4);
                    } while (_board[a, b].Value != 0);
                    ...
                }
            }
        }
    }
}
```

Listing 28: Example of a cast when making datatype changes.

In Listing 28, we can see that when calling `_rand.Next`, we need to cast the result to a `uint`, this likely increases the energy consumption of the benchmark. When looking at the change to `StringBuilder`, there is no big difference to the microbenchmark, besides being a smaller amount of `strings` that need to be concatenated. Therefore, our first change to the benchmark is changing all the cases where `uint` needs to be cast, to being `int` from the start.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Data Type 2048	282.026	8.059.475.586	157.393.821
Optimized Data Type 2048	281.382	8.158.477.796	157.013.929

Table 4.5: Table showing the elapsed time and energy measurement for the `uint` to `int` change with regards to datatypes in `2048`.

In Table 4.5 we can see an updated table with the results after changing the variables. These numbers may differ slightly from the numbers in Section 3.2.2 as other background processes may be running or the temperature is different, therefore the original `DataType` benchmark is run again to compare against.

Here we can see that there is no significant difference between the original datatype optimization, and the try at optimizing datatypes using `int` in cases where casts were needed, as the package energy consumption is within 2% of each other.

Following this, we try to switch the concatenation approach from using `StringBuilder` to using `string` interpolation, as this is the second most efficient `string` concatenation method, and might be more effective when fewer `strings` need to be concatenated.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Data Type 2048	279.719	7.994.737.305	156.090.929
Optimized Data Type 2048	283.076	8.240.086.395	157.966.797

Table 4.6: Table showing the elapsed time and energy measurement for the `string` concatenation approach change each `2048`.

In Table 4.6, we can see an updated table with the results after changing the `string` concatenation approach. Here we can see, that changing this away from a `StringBuilder` has made it perform significantly worse, as the difference is higher than 2%. Because of this, we keep the `StringBuilder` tip as something that should be utilized.

Lastly, with regards to Data Type, we try running the benchmark with the default implementation of variable datatypes, but use `StringBuilder` instead of the `string` concatenation operator (+) that is utilized in the default implementation. This is done to see if `int` is better than `uint` in a more generalized setting.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Data Type 2048	280.449	8.136.178.489	157.504.883
Optimized Data Type 2048	278.835	8.134.824.768	156.081.787

Table 4.7: Table showing the elapsed time and energy measurement for the `uint` to `int` change with `StringBuilder` in `2048`.

In Table 4.7 we can see an updated table with the results after utilizing the default variable datatypes. This shows that there is no significant difference between using `int` and `uint` for this benchmark. This is surprising, given that the analysis in [3] shows that the compiler knows a lot of ways to optimize `uint` variables when using microbenchmarks. However, as changing from `uint` to `int` and vice versa does not have a bad effect on energy consumption, we will keep it as a tip for *Energy Analyzer*. This is because, as

the microbenchmarks in [3] showed, the compiler can sometimes optimize `uint` more than `int`, so in some cases, this may be a useful tip, while in worst-case scenarios, it has no effect.

When looking at the changes to the Foreach benchmark, we see that every time a `foreach` loop is utilized, it is to iterate through a 2D array. Seeing that the difference between the default implementation and the Foreach benchmark is not significant, we consider it irrelevant whether a `for` loop or a `foreach` loop is used in this case.

When looking at the changes to the Object benchmark, we see that changing `G2048` to a `struct` made it necessary to create a default constructor for `G2048`, that is not there in the default implementation.

Because of this, we try to change the type of `G2048` to a `class` (and therefore being able to remove the constructor), while keeping `Tile` as a `struct` to see if there is a difference in performance.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Object 2048	277.932	7.785.201.063	155.170.356
Optimized Object 2048	279.274	7.939.008.758	155.904.202

Table 4.8: Table showing the elapsed time and energy measurement for the changes to object types in `2048`.

In Table 4.8 we can see the results between using the original Object benchmark, and only having `Tile` be a struct. We see in the original results that there is potential for the Object benchmark to be better than the default, therefore we will use the Object benchmark in our optimal implementation of `2048`.

When looking at the changes to the Switch benchmark, we see that all of the changes made are done where there are only two cases. As the analysis in [3] shows, the reason `switch` is more efficient than `if`, is because jump tables can be utilized after evaluation. In cases where there are only two outcomes, this is unlikely to make a difference, and therefore we consider cases where there are only two outcomes to have no significant differences between `if` and `switch`.

Lastly, we want to create the optimal version of `2048` given the results we have found. The optimal version includes the changes *Energy Analyzer* would give us. The changes we do are using `uint` instead of `int`, `struct` instead of `class`, and `StringBuilder` instead of the `string` concatenation operator (+).

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Optimal 2048	281.312	7.980.975.260	157.243.164
Default 2048	276.512	7.846.216.146	154.594.184

Table 4.9: Table showing the elapsed time and energy measurement for the "optimal" and default implementation of `2048`.

In Table 4.9 we can see that the changes to `2048` in our optimal version, are not significantly different from the default implementation, meaning that `2048` is already near-optimal compared to the changes we have evidence behind making. In fact, we can see that the measurement for the default implementation is lower than optimal in this case, however, it is within 2% and therefore not significant.

This is likely because most of the code executed in `2048` is not changed. This is unlike microbenchmarks, where we change the most important part of the benchmark every time, therefore seeing larger differences.

Furthermore, the test setup has an impact, as different test setups would have different hot paths and therefore might see more impact from the changes.

Lastly, the randomness with regards to memory can have an impact that is unknown, which makes results close to each other difficult to differentiate.

Findings

To summarize, we have the following findings/suggestions:

- `uint` and `int` have no significant difference in energy consumption for `2048`,
- `StringBuilder` is more efficient than `string` interpolation, even when only three `strings` are concatenated.
- `StringBuilder` has no significant difference to `string` concatenation operator, when only three `strings` are concatenated.
- There are no significant differences between `for` and `foreach` when iterating through 2D arrays.
- Using `struct` instead of `class` has potential to be more efficient than vice versa.

- There are no significant differences between `switch` and `if` when there are only two outcomes.
- Overall, the "optimal" implementation of `2048` does not have many changes compared to the default implementation, therefore no big changes in energy efficiency are seen.

4.2.2 21 Analysis

In Section 3.2.3, we found that making datatype changes increase the energy consumption while all other changes had no significant impact on the energy consumption. We look further into this, to see if we can figure out how to best utilize these results for *Energy Analyzer*.

When looking at the changes to datatype, we see the same as for 2048, that casts to `uint` has been necessary in some cases.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Datatype `21`	5.161	65.477.372	2.887.042
Optimized Datatype `21`	5.207	67.177.238	2.917.101

Table 4.10: Table showing the elapsed time and energy measurement for datatype changes in `21`.

In Table 4.10 we can see the results from changing the casts to `uint` to using `int` natively, while keeping the `StringBuilder` changes. The results show that having the cast is more optimal than removing it, as the changes necessary to remove the cast increases the energy consumption by more than 2%. Therefore, we use the original datatype changes for further testing.

Furthermore, when looking at the usage of `StringBuilder`, there are multiple usages of `StringBuilder`, where only two `strings` are concatenated together, which may not be optimal. Seeing as we found no significant difference between `StringBuilder` and the `string` concatenation operator (+) for 2048, we try three cases of `string` concatenation methods, for the places where only two `strings` are concatenated together. We try using `StringBuilder`, `string` interpolation, and the `string` concatenation operator (+).

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
String Interpolation Datatype 21	4.506	57.140.322	2.516.946
Datatype 21	5.176	67.904.624	2.903.444
String Concatenation Op Datatype 21	4.517	58.104.979	2.525.120

Table 4.11: Table showing the elapsed time and energy measurement for each method of `string` concatenation in 21.

In Table 4.11 we can see that using `string` interpolation or the `string` concatenation operator (+) is more efficient than using `StringBuilder` when two `strings` are concatenated. Between `string` interpolation and the `string` concatenation operator (+), there are no significant difference between them, as they are within 2% of each other, however `string` interpolation is almost significantly more efficient than the `string` concatenation operator(+). Therefore, when creating *Energy Analyzer*, when only two strings are concatenated, we advise using `string` interpolation.

When looking at `switch` we see that there is no significant difference in the result compared to the default, however, we also see in the code that there is one place where `switch` can possibly be used, however requiring a bit more work.

```
1  public class Switch {
2      public static void PlayGame() {
3          ...
4          while (playAnother) {
5              ...
6              if (total == final) {
7                  ...
8                  string choice = "n";
9                  if (choice == "y") {
10                     total = 0;
11                 }
12                 else if (choice == "n") {
13                     break;
14                 }
15                 else {
16                     Console.WriteLine("Invalid choice! Choose
                       ↪  from y or n");
17                     continue;
18                 }
```

```
19                  ...
20  }
```

Listing 29: A place where `switch` can theoretically be used instead of `if` statements.

In Listing 29, we can see a place where a `switch` could be used instead of `if`, the problem is the case where we need to break out of the `while` loop, as a `break` statement in a `switch` means you break out of the `switch`. Therefore, to get around this, while adding the least amount of overhead, we utilize `goto`.

This is done by switching the `break` with `goto`, and creating a label it can jump to after the `while` loop.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Switch 21	4.523	56.324.407	2.523.950
Optimal Switch 21	4.506	56.072.868	2.513.129

Table 4.12: Table showing the elapsed time and energy measurement for each `switch` implementation in 21.

In Table 4.12, we can see that there are no significant differences when making this change. This is likely because the amount of `if` statements that need to be evaluated is low, and therefore changes close to nothing. However, as it does not have a negative impact, and we have seen in microbenchmarks that it is optimal, we will keep this suggestion for *Energy Analyzer*.

When looking at `Exception`, there is no significant difference in the result compared to the default, this is likely because no `exception` is thrown throughout the program, and this is where most of the energy savings would be. Furthermore, the `try-catch` block is unlikely to consume a lot of energy compared to the rest of the program, as there are a lot of other things happening in the program. However, as it also does not have a negative impact, and we have seen in microbenchmarks that there is a difference, we will keep this suggestion for *Energy Analyzer*.

Lastly, we want to create the optimal version of 21 given the results we have found. We change usages of `int` to `uint` as *Energy Analyzer* would suggest, we also change `string` concatenation to `StringBuilder` when more than two strings are concatenated, however we use `string` interpolation when only two strings are concatenated. Furthermore, we change the `if`

statements to `switch` statements, including the one where we need a `goto`. Lastly, we make the changes to `exceptions`, meaning we remove the `try-catch` block, and the `throw` statements.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Optimal 21	4.475	55.999.663	2.498.010
Default 21	4.508	57.507.949	2.518.654

Table 4.13: Table showing the elapsed time and energy measurement for default and optimal version of 21.

In Table 4.13 we can see that the changes to create an optimal version of 21, have resulted in no significant change from the default. This is likely because the main changes to 21 are not a large part of the overall program. Furthermore, the test setup of 21 can have an impact on the energy consumption, as with the current setup, no `exception` is thrown in the default implementation, however, with another setup, an `exception` is thrown. As previously, there is also randomness with regards to memory that have unknown impacts, which makes the results difficult to differentiate.

Findings

To summarize, we have the following findings/suggestions:

- `String` interpolation is more efficient than `StringBuilder` when two strings are concatenated.
- When three or more `strings` are concatenated, `StringBuilder` should be used.
- There are no significant difference between `if` statements and `switch` when there are three cases.
- Using a `try-catch` block has no significant effect in 21, likely because there is a lot of code executed within the `try` block, with no `exception` being thrown.

4.2.3 99 Bottles of Beer

In Section 3.2.5, we found that making datatype changes increase the overall energy consumption while the other changes had no significant effect on the

energy consumption. We look further into this, to see if we can figure out how to utilize these results for *Energy Analyzer*.

For 99 Bottles of Beer, the only change to the datatype variant is changing `string.Format` to `StringBuilder`.

According to our microbenchmarks, `string.Format` should be the worst type of `string` concatenation, which means this is a surprising result.

```
public class DataType {
    public static uint DataType99BottlesOfBeer() {
        for (ulong i = 0; i < LoopIterations; i++) {
            ...
            for (int y = 0; y < 199; y++) {
                StringBuilder sb = new StringBuilder();
                StringBuilder sb2 = new StringBuilder();
                sb.Append(describeBottles(bottles))
                    .Append(" ")
                    .Append(Location)
                    .Append(", ")
                    .Append(describeBottles(bottles))
                    .Append(",");
                write(sb.ToString());
                ...
            }
        }
    }
}
```

Listing 30: Example of using `StringBuilder` in 99 Bottles of Beer.

In Listing 30, we can see how a `StringBuilder` is utilized.

A big difference here, to `string.Format`, is that a new object is created every time we go through the inner loop, which is likely to increase the energy consumption because of garbage collection or other memory management. This is especially true in this case, as for every iteration, 398 `StringBuilders` are created, which consumes a large amount of memory compared to the memory consumed by `string.Format`.

This was done as it is the most direct way to change a `string.Format` to a `StringBuilder`, however as `string.Format` do not create a new object every time it is used (Besides the `string`), we try to move the `StringBuilders`

outside the inner loop, and then use `StringBuilder.Clear` to clear the `StringBuilder` inside the loop.

```
public class OptimizedDataType {
    public static uint DataType99BottlesOfBeer() {
        for (ulong i = 0; i < LoopIterations; i++) {
            StringBuilder sb = new StringBuilder();
            StringBuilder sb2 = new StringBuilder();
            StringBuilder sb3 = new StringBuilder();
            ...
            for (int y = 0; y < 199; y++) {
                sb.Clear();
                sb2.Clear();
                sb.Append(describeBottles(bottles))
                    .Append(" ")
                    .Append(Location)
                    .Append(", ")
                    .Append(describeBottles(bottles))
                    .Append(",");
                write(sb.ToString());
                ...
            }
        }
    }
}
```

Listing 31: Changes to the optimized version of the datatype variant in 99 Bottles of Beer.

In Listing 31 we can see how the benchmark has changed, where the `StringBuilders` have been moved out of the loops. The reason we do not move the creation of `StringBuilders` outside the outer loop, is because we do not measure what is outside the loop, and we generally want as little code outside of that as possible.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Data Type 99 Bottles of Beer	464.223	6.389.852.818	259.349.854
Default 99 Bottles of Beer	446.442	5.965.694.336	249.352.105
Optimized Data Type 99 Bottles of Beer	404.346	5.652.342.215	225.625.698

Table 4.14: Table showing the elapsed time and energy measurement for the datatype optimized version of 99 Bottles of Beer.

In Table 4.14 we can see that this has had a significant effect on lowering the energy consumption. It had a large enough effect to show that it is better than the default implementation, meaning that `StringBuilder` is more efficient than `string.Format`, when a `StringBuilder` object has been created already.

With regards to invocation, there are no significant differences between default and the invocation variant of the benchmark. This is likely because the main part of the benchmark uses a lot of energy on concatenating `strings`. As the invocation suggestions does not increase energy consumption, we will keep the suggestion for *Energy Analyzer*.

Lastly, we want to create a theoretical optimal version with the optimized datatype variant together with the invocation variant, to see what the effect is.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Optimal 99 Bottles of Beer	390.996	5.190.948.676	217.722.005
Default 99 Bottles of Beer	434.179	5.847.384.440	241.741.048
Optimized Data Type 99 Bottles of Beer	391.158	5.248.324.490	217.672.255

Table 4.15: Table showing the elapsed time and energy measurement for the optimal variant of 99 Bottles of Beer compared to default and the optimized datatype.

In Table 4.15, we can see that the optimal version of the program uses significantly less energy than the default implementation, meaning changing from `string.Format` to `StringBuilder` is a good change when a new object does not need to be created every time it needs to be used.

Findings

To summarize, we have the following findings/suggestions:

- If a new `StringBuilder` object needs to be created often, other types of `string` concatenation may be more efficient because of memory consumption.
- `StringBuilder` is better than `string.Format` when a new `StringBuilder` object does not need to be created often.
- `Lambda expressions` with or without closure have no significant difference in energy consumption in this benchmark, likely because the main part of the benchmark consumes more energy.

4.2.4 Determine if a String has All the Same Characters Analysis

In Section 3.2.6, we found that our datatype changes had the expected effects, while the loops changes increased the energy consumption significantly.

The datatype changes were changing formatting in a `Console.WriteLine` to using a `StringBuilder` that would then create the `string` used in the `Console.WriteLine` instead.

Looking at the loops changes to the "Determine if a String has All the Same Characters", we see that we iterate through a `string` with a `foreach` loop instead of using a `for` loop.

Our experience suggests that this is because we need to use an extra variable to keep track of the index of the character, as this is used in the result.

Therefore, we find that if an index is needed, using a `for` loop is more efficient than using a `foreach` loop.

As the optimal variant is equivalent to the datatype variant, we do not create a new variant.

Findings

To summarize, we have the following findings/suggestions:

- Using `StringBuilder` is more efficient than formatting directly in a `Console.WriteLine`, as expected.
- Using a `for` loop is more efficient than using a `foreach` loop when the index of a collection/string is needed.

4.2.5 Happy Numbers Analysis

In Section 3.2.8, we found that while the datatype variant was significantly worse than the default, using the datatype changes together with the collection changes, it became better than what we would expect. We would expect the changes with collections to be slightly worse than only the collection changes, as the datatype changes by themselves were worse.

When looking at the datatype changes, two main categories of changes are made: `int` to `uint`, and changing the type of `string` concatenation to using `StringBuilder`.

In this specific case, `StringBuilder` is used to concatenate two separate `strings`, meaning, as we have found earlier, that `string` interpolation is more efficient.

However, as the `string` concatenation operator (+) is used instead, this is not the reason for the energy efficiency difference.

Therefore, the changes from using `int` to `uint` must be the reason behind the difference.

As we have previously found that changing `int` to `uint` either has no effect or decreases the energy consumption, we must look at the unique changes done in this benchmark compared to previous benchmarks.

We find that `uint` is used in `Lists`, which may be the reason behind the energy increase, as `int` may be more efficient in `Lists`.

Therefore, we change the `Lists` to using `int` to test this, however, to test this, casts are needed which may increase the energy consumption.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Data Type Happy Number	4.374	58.146.974	2.437.059
Optimized Data Type Happy Number	4.118	54.623.167	2.293.635
Default Happy Number	4.137	55.525.206	2.309.984

Table 4.16: Table showing the elapsed time and energy measurement for the datatype changes in Happy Numbers.

In Table 4.16, we can see that this has a significant effect on the result, therefore we can conclude that when using `Lists`, the developer should prioritize using `int` over `uint`.

However, as the results from the *All* variant shows, using `uint` in ordinary `arrays` is more efficient than using `int`, as that is what is used in the `Collection` benchmark.

As we found in the analysis for `21`, `string` interpolation is more efficient than `StringBuilder` when concatenating exactly two `strings`.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
All Happy Number	3.814	50.298.275	2.126.204
Optimal Happy Number	3.857	50.569.534	2.146.851
Default Happy Number	4.063	54.996.940	2.261.495

Table 4.17: Table showing the elapsed time and energy measurement for each Happy Numbers.

In Table 4.17 we can see that there are no significant difference between the *All* variant and the *Optimal* variant, therefore, we stay with the previous

suggestion of using `string` interpolation instead of `StringBuilder` when exactly two strings are concatenated.

Findings

To summarize, we have the following findings/suggestions:

- Using `int` is more efficient than `uint` in a `List`.
- Using `uint` is more efficient than `int` in an `array`.
- Using `array` is more efficient than using `List`, as expected.

4.2.6 Introspection Analysis

In Section 3.2.9 we found that there was no significant difference to the invocation part of the benchmark, which we will focus on here.

Besides this, we found that the datatype changes have no significant changes, which is likely to be because the datatype changes do not have a big enough impact compared to the rest of the code, as we have found in the earlier analysis.

When looking at the changes, we see that `reflection` is done the same amount of times in both benchmarks, while the invocation benchmark has wrapped the `reflection` invocation in a `Func` delegate.

```
1 public class Invocation {
2     public static void Main() {
3         ...
4         foreach (Assembly refAsm in
          ↪  AppDomain.CurrentDomain.GetAssemblies().Where(assembly
          ↪  => !assembly.IsDynamic)) {
5             foreach (Type type in refAsm.GetExportedTypes()) {
6                 if (type.Name == "Math") {
7                     MethodInfo? absMethod =
                      ↪  type.GetMethod("Abs", new Type[] {
                      ↪  typeof(int) });
8                     if (absMethod != null) {
9                         var absDelegate = (Func<int,
                          ↪  int>)Delegate.CreateDelegate(typeof(Func<int,
                          ↪  int>), absMethod);
```

```
10                          Console.WriteLine("bloop's abs value =
                            ↪  {0}",
                            ↪  absDelegate((int)bloopField.GetValue(null)));
11                      }
12                  }
13              }
14          }
15          ...
16      }
17  }
```

Listing 32: The `reflection` part of the invocation benchmark, where we see that the invocation is wrapped in a `Func` delegate.

In Listing 32 we can see on line 7 to 10 how `reflection` is done and how it is wrapped in a `Func` delegate.

As mentioned, as `reflection` is done every time we get to this stage, it is natural that no large difference is observed between the benchmarks.

Despite the extra overhead in creating a `Func` delegate, no increase in energy consumption is observed. Therefore, we keep the suggestion that a `reflection` invocation should be wrapped in a `delegate` when invoked.

The optimal benchmark for this benchmark is the *All* variant, as we have found no changes necessary to the suggestions based on this analysis.

Findings

To summarize, we have the following findings/suggestions:

- Despite needing `reflection` every time it is invoked, wrapping it in a `delegate` does not increase energy consumption, therefore this suggestion should not be changed.

4.2.7 World Cup Group Stage Analysis

In the results for World Cup Group Stage, we found replacing Language Integrated Query (LINQ) has a positive impact on energy consumption for the benchmark, while changes for datatypes have a negative impact on energy consumption, giving an overall positive effect on energy consumption when all changes where implemented in the same benchmark.

When analyzing the results we run the experiments multiple times again, and we now see that the `Datatype` benchmark is sometimes better than the `Default` benchmark in regards to energy consumption.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
LINQ World Cup Stage	199.510	2.805.185.372	111.129.011
All World Cup Stage	206.609	2.752.152.285	115.172.589
Default World Cup Stage	485.782	6.901.756.402	270.326.280
Data Type World Cup Stage	474.198	6.761.858.181	264.074.436

Table 4.18: Table showing the elapsed time and energy measurement for each World Cup Group Stage.

In Table 4.18 we see one of the runs where the `Datatype` benchmark is better than `Default`. This is despite the original showing the default implementation to be more efficient than the datatype implementation. Therefore, we can not conclude that the two benchmarks consume different amounts of energy. Because these results do not give further insight, we do not use them to change our suggestions.

Looking at the LINQ benchmark results, we see that our changes have made a positive impact by reducing the energy consumption to approximately half. As we did expect this result, we do not change our suggestions towards LINQ.

As we did not find any specific changes from the analysis of this benchmark, we use the results of the analysis done from previous benchmarks to make smaller changes to get the optimal version, this includes changing the `List` to using `int` instead of `uint`.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
All World Cup Stage	215.705	2.867.487.671	120.505.412
Default World Cup Stage	487.250	6.856.450.412	271.776.584
Optimal World Cup Stage	197.433	2.744.775.346	110.181.618

Table 4.19: Table showing the elapsed time and energy measurement for the optimal World Cup Group Stage benchmark compared to default and the *All* variant.

In Table 4.19 we can see that there is no significant difference between the *All* variant and the optimal variant, however they both consume significantly less energy than the default variant.

Findings

To summarize, we have the following findings/suggestions:

- The variants between benchmarks regarding datatypes is higher than expected, and we can therefore not change our suggestions regarding dataypes.
- Using LINQ is inefficient, and should be avoided.

4.2.8 Summary

In this section, we summarize the findings from the analysis of the larger benchmarks. These findings are used to be more precise when making suggestions for *Energy Analyzer*.

Findings

- Has an effect
  - Using `int` is more efficient than `uint` in a `List`.
  - Using `uint` is more efficient than `int` in an `array`.
  - `StringBuilder` is the same or more efficient than other types of concatenation when three strings or more are concatenated.
  - `String` interpolation is more efficient than `StringBuilder` when two `strings` are concatenated.
  - If a new `StringBuilder` object needs to be created often, other types of `string` concatenation may be more efficient because of memory consumption.
  - Using a `for` loop is more efficient than a `foreach` loop when the index is needed.
  - Using LINQ is inefficient, and should be avoided.
  - `Arrays` are more efficient than `Lists`.
  - Using `struct` instead of `class` has potential to be more efficient than vice versa.
- Has no effect

- `uint` and `int` have no significant difference in energy consumption in a lot of cases.
- There are no significant differences between `switch` and `if` when there are three or less outcomes.
- There are no significant differences when using `for` and `foreach` when iterating through 2D arrays.
- `Lambda expressions` with or without closure have no significant difference in energy consumption, in cases where a large amount of the energy consumption of a benchmark does not come from this.
- Despite needing `reflection` every time it is invoked, wrapping `reflection` invocation in a `delegate` does not increase energy consumption.
- Using a `try-catch` block has no significant effect when there is a lot of code executed within the `try` block, when no `exception` is thrown.

Following these findings, the formal suggestions for *Energy Analyzer* are created, seen in Section A.4.

Chapter 5

Energy Analyzer

In this chapter we design, implement, and evaluate *Energy Analyzer*.

We do this based on the suggestions in Section A.4, which are created based on the analysis in Chapter 4.

5.1 Design

This section provides the design choices and requirements of *Energy Analyzer*, which leads into the implementation of *Energy Analyzer* in Section 5.2.

5.1.1 Analyzer type

A natural way of creating a C# analyzer is creating a Roslyn analyzer, as this is what Microsoft uses for their code analysis [22]. The .NET compiler platform uses several Roslyn analyzers to inspect C# code for code style, quality, and other issues.

We look into two ways of implementing a third-party Roslyn analyzer: Integrated Development Environment (IDE) based extensions, and NuGet packages, as these are presented by Microsoft as the options of how to implement a third-party Roslyn analyzer [22].

IDE based extensions

An IDE based extension enables rules for the entire IDE that a developer can use to write higher quality code [22]. This is useful if developers have code quality rules they want to follow, or have specific analyses they want to do with their code. However, this means that all developers working

on a project need to install the same extensions, therefore this is mainly useful if only one developer is working on a project, for example for smaller programs or prototypes to test if something works or if developers use the same code quality rules across all the projects they work on.

NuGet packages

A NuGet package enables rules for the specific project, this means that any developer that works on the same project will have the same rules [22].

Choice of Extension vs NuGet Package

The main difference between using an IDE based extension and a NuGet package is the scope of the rules. As we choose to focus on the applications instead of the developers in Section 1.1, we want to create an analyzer that can be used without issue by multiple developers in a project, therefore we choose to focus on creating a NuGet package. A limitation of *Energy Analyzer* is that automatic refactoring without the user choosing to do so is not created. This has the advantage that developers know what code is used in the program and that any bugs in *Energy Analyzer* will not automatically make the analyzed program have bugs. However, it has the disadvantage that the developers need to make the changes themselves, either by using built-in code fixes in *Energy Analyzer* or by writing the code fix manually.

Analyzer Categories

It is also important to look at what category of analyzer we create. There are three categories of Roslyn analyzers: Stateless, Stateful, and Additional File analyzer.

Stateless analyzers analyze specific code units, such as a symbol, syntax node, code block, compilation, etc. independent of the state of the program [60]. Stateful analyzers analyze specific code units while keeping the state of the enclosing code block or compilation in mind [60]. Additional File analyzers read data from non-source text files included in the project [60].

Choice of Analyzer Category

As we want to create an analyzer that gives suggestions for language constructs where the state of the program sometimes has an effect, such as not

recommending `uint` if a variable is negative, a stateful analyzer is chosen.

5.1.2 Requirements

These requirements are prioritized using the MoSCoW method [61, p. 140]. The requirements are based on the knowledge gathered in the introduction, problem statement, and the design chapter of *Energy Analyzer*. They are prioritized based on how necessary they are to have a product that can be used based on the knowledge we have gathered, with less priority on bonus features.

- Must give suggestions regarding energy consumption.
- Must be created for C#.
- Must be a NuGet package.
- Must base the suggestions on the experiments done in this project and in [3], the suggestions are seen in Section A.4.
- Must give suggestions that can be used in the given situation.
- Should not have any blocking issues, i.e. the developer not being able to proceed without help, with regards to usability when tested by other developers.
- Should give an option for refactoring a given suggestion.
- Could have an easy to modify list of suggestions.
- Could provide an explanation of why this refactoring is suggested.
- Will not make changes to the code automatically without user input.

5.2 Implementation

Energy Analyzer is divided into two parts. The first part focuses on identifying the construct that should be changed and giving relevant suggestions to this construct. The second part is giving an automatic code fix for the identified code, meaning we change the developers' code to implement the suggested fix if they choose to do so.

The code fix is only provided for suggestions where it has been possible to create a code fix.

The source code for *Energy Analyzer* can be found on GitLab [2] and the package can be found on NuGet [5].

Implementing a Roslyn Analyzer

When implementing the suggestion part of a Roslyn Analyzer we extend the abstract class `DiagnosticAnalyzer` which is provided by Microsoft as an entry point for the analyzer [62].

To implement `DiagnosticAnalyzer` we are required to specify two members, the `Initialize` method and the `SupportedDiagnostics` property. `Initialize` is the method entry point called at once when registering the analyzer. When implementing the `Initialize` method we use the parameter `AnalysisContext` to register which syntax nodes to operate on. This means that we register a callback for when the analyzer reaches certain nodes. This allows us to implement callbacks that execute our analysis and display the suggestion if the analysis shows that the suggestion should be displayed.

The second member to implement from the abstract class is the property `SupportedDiagnostics` which returns a collection of the diagnostics that the analyzer is capable of producing. Furthermore, the `DiagnosticAnalyzer` attribute is added to the class, this describes which language it targets, in our case it is C#.

Implementing the code fix part of a Roslyn Analyzer uses the abstract class `CodeFixProvider`, which also contains two members which must be implemented [23].

This includes the method `RegisterCodeFixesAsync` and the property `FixableDiagnosticIds`.

The `RegisterCodeFixesAsync` method is responsible for computing the code fix itself and applying it. The `FixableDiagnosticIds` property is a collection of all the ids this code fix should show a suggestion to code fix for.

To export this code fix provider we have to annotate the class with the `ExportCodeFixProvider` attribute. This attribute describes what language the code fix is meant for and the name of the code fix.

5.2.1 Suggestions

We start by looking at giving suggestions. Some suggestions are simpler than others, therefore we look at two different suggestions, a simple and a complex one, to get an overview of how the suggestions have been implemented.

Dictionary Suggestion

The first suggestion we look at is a simple one, here we need to give the suggestion that a `Dictionary` should be used, whenever a table is used that is not a `Dictionary`.

```
1 [DiagnosticAnalyzer(LanguageNames.CSharp)]
2 public class DictionaryAnalyzer : DiagnosticAnalyzer {
3     private const string DiagnosticId = "MakeDictionary";
4     private static readonly DiagnosticDescriptor MakeDictionary
      ↪  = new(DiagnosticId, "Use Dictionary", "Using a
      ↪  Dictionary is usually more efficient than other types of
      ↪  tables.", "Usage", DiagnosticSeverity.Warning, true);
5     private static string[] RelevantTypes = { "Dictionary",
      ↪  "Hashtable", ... };
6     public override ImmutableArray<DiagnosticDescriptor>
      ↪  SupportedDiagnostics =>
      ↪  ImmutableArray.Create(MakeDictionary);
7     ...
```

Listing 33: Descriptive code used to let the IDE know what the suggestion looks like.

In Listing 33, we can see the initial descriptive code used to let the IDE know what the suggestion looks like. This is created for all of the suggestions, the most important part for the user is the `DiagnosticDescriptor`. The first parameter is a unique ID that identifies which suggestion it is, after which a title and description are given. The last two parameters are used to show what the `DiagnosticSeverity` is, which can be `Hidden`, `Info`, `Warning`, and `Error`, and whether the suggestion should be enabled by default. We have decided to give our suggestions as `Warnings` because we do not want the compiler to stop, while also ensuring that the user notices the suggestions. Besides the descriptive code, we also have a `string` array

for the types that are specifically relevant for this suggestion, this is unlike other suggestions which may not have a lot of relevant types and therefore it is unnecessary to have a helper-array for them.

```
1     ...
2     public override void Initialize(AnalysisContext context) {
3         context.ConfigureGeneratedCodeAnalysis(
          ↪  GeneratedCodeAnalysisFlags.None);
4         context.EnableConcurrentExecution();
5         context.RegisterSyntaxNodeAction(AnalyzeNode,
          ↪  SyntaxKind.LocalDeclarationStatement);
6         context.RegisterSyntaxNodeAction(AnalyzeNode,
          ↪  SyntaxKind.FieldDeclaration);
7         context.RegisterSyntaxNodeAction(AnalyzeNode,
          ↪  SyntaxKind.PropertyDeclaration);
8     }
9     ...
```

Listing 34: The `Initialize` method that is called before the code analysis is done, so we can setup everything.

In Listing 34, we can see the `Initialize` method, that is also created for all of the suggestions. This method is used to initialize different parts of the analysis, such as what the analyzer does with generated code, and whether it runs concurrently. Most importantly, here we can register methods that is called when different parts of the code is found, in this case we register the method `AnalyzeNode` whenever a `LocalDeclarationStatement`, a `FieldDeclaration`, and a `PropertyDeclaration` is found in the code. A `LocalDeclarationStatement` is a declaration of a variable done locally, while a `FieldDeclaration` is a declaration done in a `field`, and a `PropertyDeclaration` is a declaration done as a `property`.

```
1     ...
2     private void AnalyzeNode(SyntaxNodeAnalysisContext context)
      ↪  {
3         if (CanBeDictionary(context.Node)) {
4             context.ReportDiagnostic(
              ↪  Diagnostic.Create(MakeDictionary,
              ↪  context.Node.GetLocation()));
```

```
        }
    }
    ...
```

Listing 35: The `AnalyzeNode` method, which is used to give the suggestion, if the declaration should be a `Dictionary`.

In Listing 35 we can see the `AnalyzeNode` method. We call the method `CanBeDictionary` with the node to see if the declaration should be a `Dictionary`, in which case we call the `ReportDiagnostic` method with a new `Diagnostic`, which is our suggestion. This suggestion is then reported to the user at the location of the node.

```
    ...
    private static bool CanBeDictionary(SyntaxNode declaration)
    ↪ {
        string type = declaration switch {
            LocalDeclarationStatementSyntax l =>
            ↪ l.Declaration.Type.GetText().ToString().Trim(),
            FieldDeclarationSyntax f =>
            ↪ f.Declaration.Type.GetText().ToString().Trim(),
            PropertyDeclarationSyntax prop =>
            ↪ prop.Type.GetText().ToString().Trim(),
            _ => ""
        };
        return RelevantTypes.Contains(type.Split("<")[0]) &&
        ↪ !type.StartsWith("Dictionary");
    }
}
```

Listing 36: The `CanBeDictionary` method that checks if a declaration should be a `Dictionary`.

In Listing 36, we can see the `CanBeDictionary` method. Here we first determine if the declaration is a `LocalDeclarationStatementSyntax`, a `FieldDeclarationSyntax`, or a `PropertyDeclarationSyntax`, as they have no shared parent with the relevant methods. During this, we get the name

of the type of the declaration. We then return the boolean value of if it is any of the types that are relevant for this suggestion, and it is not a `Dictionary`.

StringBuilder Suggestion

The second suggestion we look at is a more complex one, where we need to give the suggestion that a `StringBuilder` should be used if three strings or more are concatenated. The initial descriptive code is similar to the `Dictionary` suggestion shown in Section 5.2.1, therefore we do not go through that in the report.

```
1 public class StringBuilderAnalyzer : DiagnosticAnalyzer {
2     ...
3     public override void Initialize(AnalysisContext context) {
4         context.ConfigureGeneratedCodeAnalysis(
          ↪ GeneratedCodeAnalysisFlags.None);
5         context.EnableConcurrentExecution();
6         context.RegisterSyntaxNodeAction(AnalyzeBlock,
          ↪ SyntaxKind.Block);
7     }
8     ...
```

Listing 37: The `Initialize` method that is called to setup the analyzer.

In Listing 37 we can see the `Initialize` method for the `StringBuilder` suggestion, the most important part is that the `RegisterSyntaxNodeAction` calls the `AnalyzeBlock` method whenever a `Block` is found in the code. We limit ourselves to only local declarations because of the complexity of the suggestion.

```
1     private void AnalyzeBlock(SyntaxNodeAnalysisContext context)
      ↪ {
2         Dictionary<string, int> count = new();
3         foreach (StatementSyntax statement in
          ↪ ((BlockSyntax)context.Node).Statements) {
4             if (statement is LocalDeclarationStatementSyntax l)
              ↪ {
5                 string type =
                  ↪ l.Declaration.Type.GetText().ToString().Trim();
```

```
6                   if (type is not "string" and not "String") {
7                       continue;
8                   }
9                   foreach (VariableDeclaratorSyntax
                    ↪  variableDeclaratorSyntax in
                    ↪  l.Declaration.Variables) {
10                      count.Add(
                        ↪  variableDeclaratorSyntax.Identifier.Text,
                        ↪  0);
11                      if (variableDeclaratorSyntax.Initializer !=
                        ↪  null) {
12                          count[variableDeclaratorSyntax
                            ↪  .Identifier.Text] +=
13                              CountRight(variableDeclaratorSyntax
                                ↪  .Initializer.Value);
14                      }
15                      else {
16                          count[variableDeclaratorSyntax
                            ↪  .Identifier.Text] = 0;
17                      }
18                  }
19              }
```

Listing 38: The `AnalyzeBlock` method that is called whenever a `Block` is encountered, 1/3.

In Listing 38, we can see the first part of the `AnalyzeBlock` method. We look through the statements in the block we look at, and see if the statement is a local declaration. If the statement is a local declaration, we look at what type it has, if the local declaration is not a `string` we go to the next statement, as no other type is relevant for this suggestion. After this, we look through the variables in the declaration and count how many concatenations are done in the initialization of the local declaration. If there is no initialization of this variable at the declaration, we set the count of it to zero.

```
1       private void AnalyzeBlock(SyntaxNodeAnalysisContext context)
        ↪  {
2           ...
```

```
3               if (statement is ExpressionStatementSyntax e &&
                ↪  e.Expression is AssignmentExpressionSyntax a &&
                ↪  a.Left is IdentifierNameSyntax i) {
4                       ...
5                       if (count.ContainsKey(i.Identifier.Text)) {
6                           count[i.Identifier.Text]++;
7                       }
8                       else {
9                           count.Add(i.Identifier.Text, 2);
10                      }
11                      if (count[i.Identifier.Text] < 3 &&
                        ↪  !a.ToString().Contains("+=")) {
12                          count[i.Identifier.Text] = 0;
13                      }
14                      else if (count[i.Identifier.Text] == 1 &&
                        ↪  a.ToString().Contains("+=")) {
15                          count[i.Identifier.Text]++;
16                      }
17                      count[i.Identifier.Text] +=
                        ↪  CountRight(a.Right);
18              }
19          }
20      ...
```

Listing 39: The `AnalyzeBlock` method that is called whenever a `Block` is encountered, 2/3.

In Listing 39, we can see the second part of the `AnalyzeBlock` method. We count the number of times a string has been concatenated using the += operator, by looking at the expression statements in the block.

If the `count` dictionary already contains the identifier, we start by adding one to the count, as we are at least doing one concatenation with this. If the `count` dictionary does not contain the identifier, we add it and set it to two, as that means the initial `string` did not have a concatenation, and it is not being concatenated with at least one other `string`, therefore two in total.

After this, we check if the expression contains the += operator, if it does not, we reset the `count` for this variable to zero if the count is below three. If the `count` for this variable is three or above, we need to send a diagnostic to the declaration of the variable, so we do not reset it. If the `count` for this

variable is one, when concatenating, we add one to the `count`, as there will always be at least two variables in the concatenation.

After this, we call the `CountRight` method to count the concatenations of the right side of the assignment.

```
1     private void AnalyzeBlock(SyntaxNodeAnalysisContext context)
      ↪ {
2         ...
3         foreach (StatementSyntax statementSyntax in
          ↪ ((BlockSyntax)context.Node).Statements) {
4             if (statementSyntax is
              ↪ LocalDeclarationStatementSyntax l) {
5                 foreach (VariableDeclaratorSyntax
                  ↪ variableDeclaratorSyntax in
                  ↪ l.Declaration.Variables) {
6                     if (count.ContainsKey(
                      ↪ variableDeclaratorSyntax.Identifier.Text)
                      ↪ && count[
                      ↪ variableDeclaratorSyntax.Identifier.Text]
                      ↪ > 2) {
7                         context.ReportDiagnostic(
                          ↪ Diagnostic.Create(MakeStringBuilder,
                          ↪ l.GetLocation()));
8                     }
9                 }
10            }
11        }
12    }
```

Listing 40: The `AnalyzeBlock` method that is called whenever a `Block` is encountered, 3/3.

In Listing 40, we can see the last part of the `AnalyzeBlock` method, we go through all of the statements in the block to find the local declarations. If the local declaration is in the `count` dictionary, and the `count` is above two, we know that at least three `strings` are concatenated with another method than `StringBuilder`, therefore we report a diagnostic at this location.

```
1     ...
```

```
    private int CountRight(ExpressionSyntax e) {
        switch (e) {
            case BinaryExpressionSyntax b:
                return CountRecursive(b);
            case InterpolatedStringExpressionSyntax i:
                return i.Contents.Count;
            case InvocationExpressionSyntax ie:
                if (ie.ToString().Contains("string.Format")) {
                    return ie.ArgumentList.Arguments.Count - 1;
                }
                else if (ie.ToString().Contains("string.Join")
                ↪  || ie.ToString().Contains("string.Concat"))
                ↪  {
                    return ie.ArgumentList.Arguments.Count;
                }
                else {
                    return 0;
                }
            default:
                return 0;
        }
    }
...
```

Listing 41: The `CountRight` method that is called from various points in the `AnalyzeBlock` method.

In Listing 41 we can see the `CountRight` method that is called at various points in the `AnalyzeBlock` method.

We can see that the expression it is called with is used in a `switch` to figure out what type of expression it is.

If the expression is a `BinaryExpressionSyntax`, we know that there are at least two parts to the expression, and therefore call the `CountRecursive` method with the expression, to count all of the parts individually. If the expression is an `InterpolatedStringExpressionSyntax`, we know the right side is an interpolated `string` and we return the number of strings in the interpolated `string`. If the expression is an `InvocationExpressionSyntax`, we know the right side is a method call, in which case we check what type of method call it is.

If the method call is to `string.Format`, we return the number of arguments to it, minus one because the first argument in that call is the overall string, not the number of strings concatenated. Otherwise, if the method call is to `string.Join` or `string.Concat`, we return the number of arguments, as these do not have extra arguments. If the method call is none of these methods or if the expression is something else than these types, we return 0 as we do not know what method it is or how many `strings` are concatenated, if any.

```
...
    private int CountRecursive(BinaryExpressionSyntax b) {
        var count = 1;
        switch (b.Left) {
            case InterpolatedStringExpressionSyntax i:
                count += i.Contents.Count;
                break;
            case InvocationExpressionSyntax ie:
                if (ie.ToString().Contains("string.Format")) {
                    count += ie.ArgumentList.Arguments.Count -
                    ↪  1;
                }
                else if (ie.ToString().Contains("string.Join")
                ↪  || ie.ToString().Contains("string.Concat"))
                ↪  {
                    count += ie.ArgumentList.Arguments.Count;
                }
                break;
            case BinaryExpressionSyntax bl:
                count += CountRecursive(bl);
                break;
            default:
                count++;
                break;
        }
...
```

Listing 42: The `CountRecursive` method that is called from the `CountRight` method, 1/2.

In Listing 42, we can see the first part of the `CountRecursive` method that is called in the `CountRight` method. We can see how the left part of the `BinaryExpressionSyntax` is handled, which is the same way as in `CountRight`, however with the left side of the `BinaryExpressionSyntax` instead of the right side.

```
...
        if (b.Right is BinaryExpressionSyntax br) {
            return count + CountRecursive(br);
        }
        switch (b.Right) {
            case InterpolatedStringExpressionSyntax i:
                return count + i.Contents.Count;
            case InvocationExpressionSyntax ie:
                if (ie.ToString().Contains("string.Format")) {
                    count += ie.ArgumentList.Arguments.Count -
                    ↪  1;
                }
                else if (ie.ToString().Contains("string.Join")
                ↪  || ie.ToString().Contains("string.Concat"))
                ↪  {
                    count += ie.ArgumentList.Arguments.Count;
                }
                return count;
            default:
                return count;
        }
    }
}
```

Listing 43: The `CountRecursive` method that is called from the `CountRight` method, 2/2.

In Listing 43, we can see the second part of the `CountRecursive` method. We handle the right side of the `BinaryExpressionSyntax`, we do this in the same way as previously, with the one exception that if the right side is still a `BinaryExpressionSyntax` we return the current count added to what the `CountRecursive` method returns.

Once all of this is done, we have the number of concatenations done for a local declaration and can report the diagnostic as shown in Listing 40.

5.2.2 Code Fixes

After looking at the suggestions, we look at how we implement code fixes for some of the suggestions. We do not implement code fixes for all of the suggestions, as some code fixes are so complex that it is not possible to figure out what the code should change into. Therefore there are some parts of *Energy Analyzer* where only a suggestion is shown, hinting that the developer should change the code themselves.

As with the suggestions, some code fixes are simpler than others, therefore we look at two different code fixes, a simple and a complex one.

Uint Code Fix

We start by looking at the code fix for `uint`, as this is a simple code fix, where we need to find the declarations where the relevant analyzer has given a suggestion, and change that to using `uint`. Besides this, the code fix also needs to remove any suffix that may appear with the number, such as `L` for `long` datatypes.

```
1 [ExportCodeFixProvider(LanguageNames.CSharp, Name =
  ↪ nameof(MakeUIntCodeFixProvider)), Shared]
2 public sealed class MakeUIntCodeFixProvider : CodeFixProvider {
3     public override ImmutableArray<string> FixableDiagnosticIds
      ↪ => ImmutableArray.Create(MakeUIntAnalyzer.DiagnosticId);
4     public override FixAllProvider GetFixAllProvider() =>
      ↪ WellKnownFixAllProviders.BatchFixer;
```

Listing 44: Initial descriptive code for the `uint` code fix.

In Listing 44 we can see the initial descriptive code for the `uint` code fix. Before declaring the class, we give the class an attribute, which describes the language the code fix is for, and the name of the code fix.

When initializing the class, we create an `ImmutableArray` consisting of the diagnostics this code fix is relevant for. After this, the last of the descriptive code is made by defining which `FixAllProvider` should be used. This makes it possible to fix multiple types of suggestions, if the code fix for the suggestion is the same. In the case of this example, it is redundant as only one `DiagnosticId` is fixable by the code fix.

```
1     public override async Task
      ↪ RegisterCodeFixesAsync(CodeFixContext context) {
2         SyntaxNode root = await
          ↪ context.Document.GetSyntaxRootAsync(
          ↪ context.CancellationToken).ConfigureAwait(false);
3         Diagnostic diagnostic = context.Diagnostics.First();
4         Microsoft.CodeAnalysis.Text.TextSpan diagnosticSpan =
          ↪ diagnostic.Location.SourceSpan;
5         LocalDeclarationStatementSyntax declaration =
          ↪ root.FindToken(diagnosticSpan.Start).Parent.
          ↪ AncestorsAndSelf().OfType<
          ↪ LocalDeclarationStatementSyntax>().FirstOrDefault();
6         ...
```

Listing 45: The `RegisterCodeFixesAsync` method that is used to give the codefix to the developer 1/2.

In Listing 45, we can see where we begin creating the code fix. We start by getting the root of the document we want to change, meaning the node that contains all of the code. After this, we get the first diagnostic and the span of the code where this diagnostic is active. This is received from the analyzer that is relevant to this code fix, in this case, `MakeUIntAnalyzer`. After this we get the declaration that we want to change, we look for `LocalDeclarationSyntax`, `PropertyDeclarationSyntax`, and `FieldDeclarationSyntax`, because these are the three types of declarations that exist.

```
1         if (declaration != null) {
2             CodeAction action = CodeAction.Create(
3                 "Make uint",
4                 c => MakeUIntLocal(context.Document,
                  ↪ declaration, c),
5                 nameof(MakeUIntCodeFixProvider));
6             context.RegisterCodeFix(action, diagnostic);
7         }
8         ...
9     }
```

Listing 46: The `RegisterCodeFixesAsync` method that is used to give the codefix to the developer 2/2.

In Listing 46 we can see the second part of the `RegisterCodeFixsAsync` method, we check which of the declarations are not `null` and create a relevant `CodeAction` with the relevant method, after which the code fix is registered. A `CodeAction` is the action that will occur if the developer chooses to use the code fix. The registered code fixes are the list of code fixes that are provided to the user. The methods called with regards to the declarations are almost the same, with the only difference being the type of declaration, therefore we only look through the method for `LocalDeclarationSyntax` in this section.

```
1     private static async Task<Document> MakeUIntLocal(Document
      ↪ document, LocalDeclarationStatementSyntax
      ↪ localDeclaration, CancellationToken cancellationToken) {
2         VariableDeclarationSyntax variableDeclaration =
          ↪ localDeclaration.Declaration;
3         TypeSyntax variableTypeName = variableDeclaration.Type;
4         TypeSyntax typeName =
          ↪ SyntaxFactory.ParseTypeName("uint")
          ↪ .WithLeadingTrivia(variableTypeName
          ↪ .GetLeadingTrivia())
          ↪ .WithTrailingTrivia(variableTypeName
          ↪ .GetTrailingTrivia());
5         var newVariableDeclarators = new
          ↪ SeparatedSyntaxList<VariableDeclaratorSyntax>();
6         foreach (VariableDeclaratorSyntax variableDeclarator in
          ↪ variableDeclaration.Variables) {
7             VariableDeclaratorSyntax fix =
8                 variableDeclarator.
                  ↪ WithInitializer(FixVariableDeclarator(
                  ↪ variableDeclarator.Initializer));
9             newVariableDeclarators =
              ↪ newVariableDeclarators.Add(fix);
10        }
```

Listing 47: The `MakeUIntLocal` method which is used to change the code 1/2.

In Listing 47 we can see the `MakeUIntLocal` method. We start by getting the declaration of the `LocalDeclaration`, the type of it, and the trivia around it. Trivia refers to Syntax Trivia, which includes elements such as comments, preprocessor directives, and various formatting elements such as spaces and newlines [63]. The trivia is applied to the new type of the declaration we create. After this, we go through all of the variables initialized in this declaration, as there can be multiple. When doing this, we look through the initializers, as they could have suffixes for another datatype which would not be legal when changed to `uint`. We fix the initializers by calling the `FixVariableDeclarator` method with the initializer of the declaration.

```
1         TypeSyntax simplifiedTypeName =
          ↪ typeName.WithAdditionalAnnotations(Simplifier.Annotation);
2         variableDeclaration =
          ↪ variableDeclaration.WithType(simplifiedTypeName).
          ↪ WithVariables(newVariableDeclarators);
3         LocalDeclarationStatementSyntax newLocal =
          ↪ localDeclaration.WithDeclaration(variableDeclaration);
4         SyntaxNode root = await
          ↪ document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
5         SyntaxNode newRoot = root.ReplaceNode(localDeclaration,
          ↪ newLocal);
6         return document.WithSyntaxRoot(newRoot);
7     }
```

Listing 48: The `MakeUIntLocal` method which is used to change the code 2/2.

In Listing 48 we can see the second part of the `MakeUIntLocal` method. Here we simplify the typename of the variable, if it can be simplified further than it is, and replace the type of the variable declaration with the simplified type. We also replace the variable initializers with the new ones that were fixed using the `FixVariableDeclarator` method. After this, we create the new `LocalDeclarationStatementSyntax` which contains the declaration that we have created throughout the method, after which we get the root of the document and replace the old `LocalDeclarationStatementSyntax` node with the new `LocalDeclarationStatementSyntax` node. In the end, we re-

turn the new document with the new node, and the code fix will have been applied to the document.

```
1     private static EqualsValueClauseSyntax?
      ↪ FixVariableDeclarator(EqualsValueClauseSyntax?
      ↪ initializer) {
2         if (initializer == null) {
3             return initializer;
4         }
5         return initializer.WithValue(
          ↪ SyntaxFactory.ParseExpression(Regex.Replace(
          ↪ initializer.Value.ToString(), "([L|l|u|U|ul|UL|])",
          ↪ "")));
6     }
7 }
```

Listing 49: The `FixVariableDeclarator` method that fixes the `initializer` of the `declaration`.

In Listing 49 we can see the `FixVariableDeclarator` method that is used to replace the suffix of a variable, if it exists. We use `regex` to replace any occurrence of a valid suffix with no suffix, as no suffix is needed when `uint` is provided explicitly in the variable declaration. We then return the new initializer with the replaced suffix, so no suffix is used in the variable declaration.

Lambda Expression Code Fix

We look at the code fix for lambda expressions, where we change lambda expressions with closure to lambda expressions with parameters so that the lambda expressions will not access any variables outside the lambda expression itself.

The complex part of this code fix is to find the type of the variables that need to be changed, as well as finding the places in the expression where a type needs to be inserted and a variable identifier needs to be inserted.

Furthermore, we need to ensure that we do not add too many or too few of the types to the generic type of the lambda expression, as the code would not be able to compile in that case.

The initial descriptive code for the lambda expression code fix is similar to the uint code fix, which means we will not describe that in the report.

```
[ExportCodeFixProvider(LanguageNames.CSharp, Name =
    nameof(ReflectionCodeFixProvider)), Shared]
public class LambdaCodeFixProvider : CodeFixProvider {
    ...
    private static async Task<Document>
        MakeLambdaWithoutClosure(CodeFixContext context,
        CancellationToken c) {
        ...
        ParenthesizedLambdaExpressionSyntax lambdaExpression =
            root!.FindToken(diagnosticSpan.Start).Parent!.
            AncestorsAndSelf().OfType<
            ParenthesizedLambdaExpressionSyntax>().FirstOrDefault()!;
        VariableDeclarationSyntax? ancestor;
        if (lambdaExpression.Parent?.Parent?.Parent is
            VariableDeclarationSyntax v) {
            ancestor = v;
        }
        ...
```

Listing 50: Finding the lambda expression and the declaration of the lambda expression.

In Listing 50 we can see the initial code used to find the old lambda expression that needs to be updated.

Whenever we add a parameter to the lambda expression, we also need to add the type of the parameter to the declaration of the lambda expression.

```
        GenericNameSyntax? genericType = ancestor.Type as
            GenericNameSyntax;
        GenericNameSyntax newGenericType = genericType!;
        newGenericType = newGenericType.WithTypeArgumentList(
            newGenericType.TypeArgumentList.WithArguments(
            newGenericType.TypeArgumentList.Arguments.RemoveAt(
            newGenericType.TypeArgumentList.Arguments.Count -
            1)));
        ParenthesizedLambdaExpressionSyntax? newLambdaExpression
            = lambdaExpression;
```

```
5           foreach (IdentifierNameSyntax identifierNameSyntax in
            ↪ LambdaAnalyzerV2.MapFromLambdaToIdentifiersToFix[
            ↪ lambdaExpression.ToString()]) {
6               string variableName =
                ↪ identifierNameSyntax.Identifier.Text;
7               ILocalSymbol? symbol =
                ↪ document.GetSemanticModelAsync().Result!
                ↪ .LookupSymbols(diagnostic.Location.SourceSpan.Start)
                ↪ .First(a => a.Name == variableName) as
                ↪ ILocalSymbol;
8               ParameterSyntax parameter =
                ↪ SyntaxFactory.Parameter(...,
                ↪ SyntaxFactory.Identifier(variableName), ...);
9               string typeName = symbol!.Type.ToString()!;
```

Listing 51: Initializing the new generic type of the lambda expression and the new lambda expression, as well as finding the variable name and symbol, and at the end creating a parameter and finding the type.

In Listing 51 we can see how we find the type of the declaration of `newLambdaExpression`. This is the type we will update with the type of the parameters. To ensure that the return type is at the end of the list of types in the declaration, we remove it from the initial list of types, and add it after all the other types are added. We do this because the last type in the list is the return type of the lambda expression.

We initialize `newLambdaExpression`, so we can change it with the changes needed.

In the `foreach` loop we go through all of the variables that have been found by the analyzer, that need to be moved to a parameter. We find the name of the variable, and find the symbol that is relevant to it.

We create a parameter with the name of the variable and find the type of the symbol, so we can input these into the parameter list of the lambda expression, and the type list of the declaration respectively.

```
1               newGenericType = newGenericType.
                ↪ AddTypeArgumentListArguments(SyntaxFactory
                ↪ .ParseTypeName(typeName));
2               newLambdaExpression =
                ↪ SyntaxFactory.ParenthesizedLambdaExpression()
```

```
3               ...
4               .WithParameterList(newLambdaExpression.ParameterList
                ↪  .AddParameters(parameter))
5               ...
6           }
7           newGenericType =
            ↪  newGenericType.AddTypeArgumentListArguments(
            ↪  SyntaxFactory.ParseTypeName(genericType!.TypeArgumentList
            ↪  .Arguments[^1].ToString()));
8           Dictionary<SyntaxNode, SyntaxNode> replacements = new
            ↪  Dictionary<SyntaxNode, SyntaxNode>();
9           replacements.Add(genericType!, newGenericType!);
10          replacements.Add(lambdaExpression,
            ↪  newLambdaExpression!);
11          root = root.ReplaceNodes(new List<SyntaxNode> {
            ↪  genericType!, lambdaExpression }, (oldNode, _) =>
            ↪  replacements[oldNode].WithTriviaFrom(oldNode));
12          return document.WithSyntaxRoot(root);
13      }
14  }
```

Listing 52: Creating the new generic type and new lambda expression, inserting them into the replacements to be made in the code, and creating the new document.

In Listing 52 we can see how the `newGenericType` is updated with the type of the new parameter. Next the `newLambdaExpression` is updated with the new parameter, and then the `foreach` loop continues until all the parameters have been added.

After the `foreach` loop is done, we add the returntype of the `newLambdaExpression` back to the list of types in the declaration of the `newLambdaExpression`.

After this, we create a `Dictionary` of the replacements we want to make to the code. We add the type of the declaration of the `newLambdaExpression`, as well as the `newLambdaExpression`.

Lastly, we replace the old nodes with the new ones in the root node and return the new document.

5.3 Evaluation

To determine how well *Energy Analyzer* works, and how easy it is to use, we do two types of evaluation.

We use *Energy Analyzer* on larger projects to determine how much energy *Energy Analyzer* saves as our first type of evaluation.

The second type of evaluation is usability tests, where another group of developers use *Energy Analyzer* to see how easy it is to use.

5.3.1 Energy Evaluation

We use the tool on the source code of two large programs: CUP [64] and SLY [65].

We will have three measurements:

- the default implementation,
- using only the changes provided by the code fixes, and
- all the suggestions given by *Energy Analyzer* being addressed.

As *Energy Analyzer* provides both suggestions and code fixes, the second measurement does not address all the issues presented by *Energy Analyzer*, therefore we have the third measurement where we manually address the remaining suggestions. The changes to each of the programs can be seen in the *3rd Party Examples Projects* folder on our Gitlab repository [2].

The p-values for the results can be seen in Section A.5.

As we evaluate the benchmarks instead of using them to determine future suggestions, we only look through the tables with results to see how well our suggestions have worked.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Original	3.003	42.385.327	1.691.376	N/A
Code Fixes	2.516	36.006.834	1.413.389	-15,05%
All	2.530	36.072.308	1.419.435	-14,89%

Table 5.1: Table showing the elapsed time and energy measurement for each CUP, the difference from default is with regards to package energy.

In Table 5.1 we can see the results from evaluating the CUP [64] program. Our code fixes have cut down on energy consumption by 15%. All suggestions being addressed does not cut further down on energy consumption,

however, our experience suggests that the changes that were made, which were not already dealt with by the code fix providers, were not a large part of the code path or were too small to give a significant difference. Another explanation could be that the combination of the suggestions does not provide the same energy savings as individual savings does, this is discussed in Section 8.1.2.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Difference from Default
Original	4.631.859	63.975.830.529	2.685.393.029	N/A
Code Fixes	4.648.180	64.191.706.597	2.682.202.257	0,33%
All	4.670.993	64.332.241.757	2.691.817.790	0,56%

Table 5.2: Table showing the elapsed time and energy measurement for each Sly, the difference from default is with regards to package energy.

In Table 5.2 we can see the results from evaluating the Sly [65] program. The changes we have made with help from *Energy Analyzer* have not changed the energy consumption significantly. Based on our experience this is because the changes made are not a significant part of the code path when running the benchmark we have set up, and therefore have minimal effect on the results.

From these results, we can see that there are potential energy savings from using *Energy Analyzer*, and in the worst case, no significant difference is observed.

5.3.2 Usability Test

To test how usable *Energy Analyzer* is, we perform usability tests with software developers.

We do this by asking a group of developers to use *Energy Analyzer* on a program we have created and giving them instructions on what to do, so we can observe any issues they may have.

If any issue is blocking, we need to make changes to *Energy Analyzer* until it is usable, per our requirement "Should not have any blocking issues, i.e. the developer not being able to proceed without help, with regards to usability when tested by other developers." seen in Section 5.1.2.

Test Setup

The usability tests consist of five tasks a test subject, i.e. a developer, should complete during the test to see if there are any usability issues, these tasks

can be seen in Section A.6.

The usability test is conducted with three student developers that have multiple years of experience in C#, and are therefore part of our target group.

Ideally, five developers should conduct the usability test, as they find approximately 80% of usability issues, and the amount of usability issues found does not increase significantly beyond five subjects [66]. However, we conduct the usability tests with only three developers due to time constraints.

Besides selecting the developers to conduct the usability test with, we also need a way to categorize the usability issues into severity categories to prioritize how we should fix them.

We use four categories of severity: Cosmetic, Severe, Critical, and External, inspired by [67].

The severity levels are based on how long it takes to complete a task, how irritated the developer gets during the task, and what the result is compared to the expected result.

- Cosmetic: if it causes little to no irritation, or there are small differences between expected and actual results.
- Severe: if it causes medium amounts of irritation, or there are some differences between expected and actual results.
- Critical: if it is not solved, causes large amounts of irritation, or there are large differences between expected and actual results.
- External: if the issue is regarding a system outside our control, such as the IDE or the Package Manager.

Test Results

During the usability test, we log what issues appear during the tests.

ID	Usability Problem	Severity	Subject(s)
1	Counted suggestions that are not from *Energy Analyzer*.	External	1, 2
2	Initially missed the `struct` suggestion.	Cosmetic	1, 2, 3
3	Slight confusion at the `switch` suggestion spanning the entire `if` statement instead of only the keyword.	Cosmetic	1, 3
4	Initially missed new suggestions appearing after code fixing an old suggestion.	Cosmetic	1, 2
5	Did not notice which suggestions and code fixes were from *Energy Analyzer* versus other plugins.	Cosmetic	1, 2, 3
6	Used some time looking for the name of *Energy Analyzer* when installing NuGet package.	Cosmetic	2
7	Used a different code fix than one provided by *Energy Analyzer* which created an error.	External	2
8	Slight confusion at what to do with the `try-catch` suggestion, but satisfied with the suggestion.	Cosmetic	3

Table 5.3: Usability issues with the severity and which subjects had the issue.

In Table 5.3 we can see the usability issues with an associated ID, a severity level, and which subjects had the issue.

We can see that there are only cosmetic or external issues with *Energy Analyzer*, with none of the cosmetic issues blocking the usage of *Energy Analyzer*. None of the issues created significant differences from the expected result and caused any irritation, except for the usability problem with id 7. This problem is considered an external issue because the error occurred because of a code fix not provided by *Energy Analyzer*, furthermore, the issue created an error that would be fixed by removing one line of code, meaning it is an error that is easy to fix.

The issue with id 3 is fixed by moving the suggestion to the first `if` keyword in the statement. The other cosmetic issues are not simple to fix and therefore will be left for future work.

All of the developers were satisfied with the usability of *Energy Analyzer*, meaning we consider it to have satisfied the requirement.

Chapter 6

Reflections

In this chapter we reflect on the project, to give insight into what went right and what could be better in the future. We start by reflecting on our choice of benchmarks. We then reflect on our implementation of *Energy Analyzer*. Lastly, we reflect on our work process.

6.1 Benchmarks

We chose to research microbenchmarks and larger benchmarks to get insight into how different language constructs affect programs. This has been a good choice because we gathered information that was used for our linter: *Energy Analyzer*.

Getting extra information within `Lambda Expressions` and `Exceptions`, as well as improving our methodology for evaluation and analysis for benchmarks has been a good choice, as that improves the suggestions in *Energy Analyzer*. Knowing the reasons behind the results means we can understand cases where the results may differ from the expected, and thereby take this into account when creating the suggestions for *Energy Analyzer*.

Generalizing the microbenchmarks to larger benchmarks has also been a good choice, as this has given extra insight into how language constructs work in larger contexts. We have found that there are cases where results from microbenchmarks are not directly applicable to larger programs. The knowledge found from this has been used to improve the suggestions in *Energy Analyzer*. The larger benchmarks chosen did give us insight into a large amount of the language constructs we had, therefore they were a good choice, however there were still some language constructs that are yet to be

tested in larger benchmarks making this a possible place for improvement.

6.2 Energy Analyzer

We have implemented a linter called *Energy Analyzer*. This analyzer has shown to improve energy consumption by up to 15% in one benchmark, while another benchmark has shown no energy difference.

We consider *Energy Analyzer* to be easy to use and well implemented. Making usability tests for *Energy Analyzer* has been a good choice, as that gathered insight into how other developers would use *Energy Analyzer* and if there were any major usability issues.

There were no major usability issues within our test group of three people when using *Energy Analyzer*, which suggests that we have created a usable tool.

We were surprised at how simple it was to get started with creating a Roslyn Analyzer, however, we were also surprised at how fragile the code was when creating a linter, as we encountered many bugs in our code. We mitigated a lot of the bugs by utilizing test-driven development [68], which was a good choice as our experience shows that we otherwise would have spent months debugging our project, instead of the two weeks it took. We recommend ourselves and future developers of linters to use more time initially to research the APIs and workflow of creating linters, as our experience shows that this helps lessen the time it takes to debug and fix bugs.

6.3 Work Process

Our work process has worked well to keep a structure in the project, with regards to how the work has been divided and how well we were prepared for unforeseen consequences.

This semester we were a smaller group of three instead of six people, which created some worry at the start of the project. Having fewer people working on the project than in previous semesters meant we had to divide the workload differently, which we did successfully. This is because of our work process, as that has made it possible to keep track of everything, including being able to keep track of tasks that need to be done, if any difficulties occurred or if things went smoother than expected.

Creating tasks that need to be done was simple, because we were three people and that made discussions about these tasks shorter than in previous projects with larger groups. Furthermore, having these tasks made it easy to continue work, as we would just tell the other two people that we are done with a task and then continue with another task without much discussion. We used Notion [25] to keep track of the tasks, and how far we were in the project, which we consider a good choice as that helped us keep an overview.

Chapter 7

Conclusion

In this project, we expand on the research into the energy efficiency of C# language constructs and how they affect the energy usage of programs [3]. We use this knowledge to create a NuGet package consisting of a linter called `Energy Analyzer` [5], which is a Roslyn analyzer that helps developers by giving suggestions and code fixes to reduce the energy consumption of their programs.

Overall we have three main contributions in this project:

1. Results from microbenchmarks regarding `lambda` expressions and `exceptions`,
2. results from generalizing the results from microbenchmarks to larger benchmarks, and
3. a linter named *Energy Analyzer* that serves to decrease the energy consumption of C# programs.

We improve upon our earlier method [3, p.60, p.111] in researching C# language constructs using microbenchmarks, by adding a few steps into our procedure of analyzing and presenting the results. These extra steps help sanity check if our results are plausible, and help analyze the results to understand the results.

Besides the expanded research in microbenchmarks, we use our results from the microbenchmarks in larger benchmarks to see if the results can be generalized into bigger programs. This is done to understand what language constructs are the most efficient and where they might not be as efficient as shown in the microbenchmarks. We found some interesting results such as `string` interpolation is the most efficient type of concatena-

tion when only two strings are concatenated, and that `for` loops are more efficient than `foreach` loops when the index of an `array` is used.

Following the larger benchmarks, we create `Energy Analyzer`, which gives suggestions and code fixes based on the results found. Creating *Energy Analyzer* was a learning experience, as we have no previous experience creating linters. We implement a small part of *Energy Analyzer*, ensuring it works before implementing the next part of *Energy Analyzer*, which helps us understand how linters work.

We evaluate *Energy Analyzer* using both usability tests and by evaluating the performance of larger programs. The usability tests with three developers, only find cosmetic issues and suggest that `Energy Analyzer` is easy to use. We find that applying the code fixes lowers the energy consumption by up to 15,05% energy in one benchmark, while another benchmark gets no significant improvement.

Chapter 8

Future Work

In this chapter, we provide suggestions for future work with regards to the benchmarks, analysis, and *Energy Analyzer*.

8.1 Benchmarks

There is more work to be done with regards to benchmarks of language constructs. Specifically, more categories can be tested and in a different way, as well as more analysis that can be done to understand the results of the benchmarks better.

8.1.1 Categories

There are categories of language constructs that we have not tested, for example, different ways of dealing with concurrency. Besides this, we have found different results depending on the larger benchmarks, which means multiple setups of each language construct could be created. For example testing `try-catch` blocks versus checking for two or more things in an `if` statement, or seeing the difference between a `switch` with 4 cases and an `if` with 4 cases, using `int` instead of `uint` in a `Dictionary` etc.

Additional larger benchmarks could be tested to get a broader overview of whether the effects found in microbenchmarks hold true when generalized. For example, it could be useful to test a larger benchmark with boxed datatypes, to see what the effect is compared to the results from microbenchmarks in [3].

8.1.2 Analysis

We found in Section 4.1.1 that garbage collection affects the results found in benchmarks, therefore looking into this further would help create better benchmarks and gather more accurate results.

Furthermore, analyzing if the results are the same on multiple computers, running different operating systems and with different processors, etc. could give insight into whether the results found can be used in all cases or if any results should be dismissed because it is not always clear-cut.

Figuring out if significant results can be found by running the benchmarks more times would be useful, as we currently have the limitation that if two benchmarks are within 2% of each other they are not significantly different. This limitation is based on experience from previous research [3], where any benchmark that is less than 2% more efficient than another benchmark could be less efficient the next time the benchmarks were run.

Lastly, it would be interesting to figure out if the energy savings always are cumulative or if they have different effects when used together in different ways. An example we found is that using `uint` in a `List` is less efficient than using `int` in a `List`, which was contrary to our initial suggestions.

8.2 Energy Analyzer

With regards to our tool *Energy Analyzer*, different improvements can be made, for example improving the suggestions to have a better understanding of the code structure, implementing complex code fixes, creating *Energy Analyzer* as an Integrated Development Environment (IDE) extension instead of only a NuGet package, and utilizing semantic analysis.

8.2.1 Suggestions

If more research is done into benchmarks, the suggestions in *Energy Analyzer* should be adjusted accordingly. Furthermore, there are cases currently where suggestions are made despite giving errors in the program, this is not trivial to fix. An example is `uint` being suggested despite the variable becoming negative at a later point in the program.

The errors can be mitigated by trying to create a program with the suggestion implemented, seeing if there is an error and in that case, dismiss

the suggestion. This would not be able to remove all errors that could occur after implementing a suggestion, however, it could mitigate some of them.

Besides this, an easier way to implement suggestions without having an understanding of code analysis could be interesting to work on. This could for example be by setting up a text file that has suggestion descriptions, language constructs to look for, and what to replace them with. However, ensuring that these suggestions are always placed correctly is difficult.

8.2.2 Code Fixes

There are currently some code fixes that are not implemented, despite a formal suggestion existing for it. This is because there are parts of the code fixes that need to be implemented that are complex and are difficult to ensure that no error is made in the code fix. Like with the suggestions, the code fix could be applied and we can check if there is an error, and in that case not give the code fix.

Furthermore, implementing code fixes for additional suggestions found in the future is also important, to ensure that *Energy Analyzer* is up-to-date with the latest research.

8.2.3 IDE Extension

In the future, *Energy Analyzer* could be both a NuGet package and an IDE extension to cover more use cases for developers. Microsoft has documentation [22] for creating a Roslyn analyzer as a NuGet package and an IDE extension, therefore we believe that this is possible to do.

This would make it possible for developers to have *Energy Analyzer* analyze all the projects they work on, instead of having to install *Energy Analyzer* to all the projects they want analyzed.

8.2.4 Semantic Analysis

Creating semantic analysis in *Energy Analyzer* could be useful to figure out what language construct should be used depending on the code flow. A step in this direction could be using Aspect-Oriented Programming [69] to analyze the workflow of programs. An example where semantic analysis could be an improvement, is analyzing if a program would be correct after changing a `int` variable to a `uint` variable, as our current way of analyzing

does not ensure that the `int` never becomes negative, and therefore changing it to `uint` would break the program.

8.3 More Tools

More tools can be made that can utilize the research done in this project and in [3].

For example, a tool that automatically makes changes to a program without developer input, to ensure that the most efficient language constructs are utilized.

Another tool could be one that automatically evaluates how efficient a program could be if the code fixes from *Energy Analyzer* are implemented. This tool could be used as a test to see if a developer should bother implementing the code fixes from *Energy Analyzer* in the places where possible.

Besides this, a tool that can automatically evaluate any program with regards to energy consumption could be useful. The program can build upon the research done in [3], specifically the framework `CSharpRapl` found on GitLab [4].

Bibliography

[1] Lasse Stig Emil Rasmussen, Milton Kristian Lindof, and Søren Bech Christensen. *Benchmarks-P10*. URL: https://gitlab.com/ImDreamer/benchmarks-p10.

[2] Søren Bech Christensen Lasse Stig Emil Rasmussen Milton Kristian Lindof. *EnergyAnalyzer*. URL: https://gitlab.com/ImDreamer/analyzer-p10/-/tree/main (visited on 2022-05-10).

[3] Aleksander Øster Nielsen, Kasper Jepsen, Lasse Stig Emil Rasmussen, Milton Kristian Lindof, Rasmus Smit Lindholt, and Søren Bech Christensen. *Benchmarking C# for Energy Consumption*. Aalborg University, 2022.

[4] Aleksander Øster Nielsen, Kasper Jepsen, Lasse Stig Emil Rasmussen, Milton Kristian Lindof, Rasmus Smit Lindholt, and Søren Bech Christensen. *CsharpRAPL*. URL: https://gitlab.com/ImDreamer/CsharpRAPL (visited on 2022-05-23).

[5] Søren Bech Christensen Lasse Stig Emil Rasmussen Milton Kristian Lindof. *Analyzer-P10*. URL: https://www.nuget.org/packages/EnergyAnalyzer/ (visited on 2022-05-10).

[6] Aleksander Øster Nielsen, Kasper Jepsen, Lasse Stig Emil Rasmussen, Milton Kristian Lindof, Rasmus Smit Lindholt, and Søren Bech Christensen. *CsharpRAPL*. URL: https://www.nuget.org/packages/CsharpRAPL/ (visited on 2022-06-08).

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] Mohit Kumar, Xingzhou Zhang, Liangkai Liu, Yifan Wang, and Weisong Shi. "Energy-Efficient Machine Learning on the Edges". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2020, pp. 912–921.

[18] TIOBE. *TIOBE Index for August 2021*. 2021-08. URL: https://www.tiobe.com/tiobe-index/ (visited on 2022-02-09).

[19] Testim. *What Is a Linter? Here's a Definition and Quick-Start Guide*. 2021-06-18. URL: https://www.testim.io/blog/what-is-a-linter-heres-a-definition-and-quick-start-guide/ (visited on 2022-03-04).

[20] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. "What do programmers know about the energy consumption of software?" In: *PeerJ PrePrints* 3 (2015), e886v2.

[21] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. "An empirical study of practitioners' perspectives on green software engineering". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 237–248.

[22] Microsoft. *Overview of source code analysis*. URL: https://docs.microsoft.com/en-us/visualstudio/code-quality/roslyn-analyzers-overview?view=vs-2022 (visited on 2022-02-23).

[23] Microsoft. *Tutorial: Write your first analyzer and code fix*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/tutorials/how-to-write-csharp-analyzer-code-fix#write-the-code-fix (visited on 2022-05-06).

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

[25] Inc Notion Labs. *Notion*. 2022. URL: https://www.notion.so (visited on 2022-02-01).

[26] GitLab. *GitLab*. URL: https://gitlab.com/ (visited on 2022-02-01).

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

[28] Guang Wei, Depei Qian, Hailong Yang, Zhongzhi Luan, and Lin Wang. "FPowerTool: A Function-Level Power Profiling Tool". In: *IEEE Access* 7 (2019), pp. 185710–185719. DOI: 10.1109/ACCESS.2019.2961507.

[29] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. "Estimating mobile application energy consumption using program analysis". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 92–101. DOI: 10.1109/ICSE.2013.6606555.

[30] 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.

[31] 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.

[32] 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 2022-02-09).

[33] Peter Sestoft. "Microbenchmarks in Java and C#". In: *Lecture Notes, September* (2013).

[34] Stan Brown. "How big a sample do I need?" In: *Brownmath* (2013).

[35] 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.

[36] 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 2022-02-09).

[37] Dr. Saul McLeod. *What a p-value tells you about statistical significance*. 2019. URL: https://www.simplypsychology.org/p-value.html (visited on 2022-02-09).

[38] 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 2022-02-09).

[39] Rosetta Code. *Rosetta Code*. URL: http://rosettacode.org/wiki/Rosetta_Code (visited on 2022-02-28).

[40] The Computer Language Benchmarks Game. *The Computer Language Benchmarks Game*. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html (visited on 2022-03-03).

[41] Rosetta Code. *2048*. URL: http://rosettacode.org/wiki/2048 (visited on 2022-02-28).

[42] Rosetta Code. *21 game*. URL: http://rosettacode.org/wiki/21_game (visited on 2022-02-28).

[43] Rosetta Code. *4-rings or 4-squares puzzle*. URL: http://rosettacode.org/wiki/4-rings_or_4-squares_puzzle (visited on 2022-02-28).

[44] Rosetta Code. *99 bottles of beer*. URL: http://rosettacode.org/wiki/99_bottles_of_beer (visited on 2022-02-28).

[45] Rosetta Code. *Determine if a string has all the same characters*. URL: http://rosettacode.org/wiki/Determine_if_a_string_has_all_the_same_characters (visited on 2022-02-28).

[46] Rosetta Code. *Dijkstra's algorithm*. URL: http://rosettacode.org/wiki/Dijkstra's_algorithm (visited on 2022-02-28).

[47] Rosetta Code. *Happy numbers*. URL: http://rosettacode.org/wiki/Happy_numbers (visited on 2022-02-28).

[48] Rosetta Code. *Introspection*. URL: https://rosettacode.org/wiki/Introspection (visited on 2022-03-01).

[49] Rosetta Code. *World Cup group stage*. URL: http://rosettacode.org/wiki/World_Cup_group_stage (visited on 2022-02-28).

[50] Bartosz and Yoh Deadfall. *PowerUp*. URL: https://rat.dev/badamczewski/PowerUp.

[51] Microsoft. *Func<T,TResult> Delegate*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.func-2?view=net-6.0 (visited on 2022-02-14).

[52] Microsoft. *Action<T> Delegate*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.action-1?view=net-6.0 (visited on 2022-02-14).

[53] Microsoft. *Delegates (C# Programming Guide)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/ (visited on 2022-02-14).

[54] Microsoft. *GC.TryStartNoGCRegion Method*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.gc.trystartnogcregion?redirectedfrom=MSDN&view=net-6.0#overloads (visited on 2022-02-15).

[55] Microsoft. *GC.EndNoGCRegion Method*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.gc.endnogcregion?redirectedfrom=MSDN&view=net-6.0#System_GC_EndNoGCRegion (visited on 2022-02-15).

[56] Microsoft. *Lambda expressions (C# reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions (visited on 2022-02-14).

[57] Microsoft. *Exception Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.exception?view=net-6.0 (visited on 2021-02-14).

[58] Microsoft. *ArgumentException Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.argumentexception?view=net-6.0 (visited on 2021-02-14).

[59] Microsoft. *DivideByZeroException Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.dividebyzeroexception?view=net-6.0 (visited on 2021-02-14).

[60] Manish Vasani, Tom Meschter, Leonid Tsarev, and Allison Chou. *Analyzer Samples.md*. URL: https://github.com/dotnet/roslyn/blob/main/docs/analyzers/Analyzer%20Samples.md (visited on 2022-03-22).

[61] David Benyon. *Designing interactive systems: A comprehensive guide to HCI, UX and interaction design*. Pearson Edinburgh, 2014. ISBN: 978-1-4479-2011-3.

[62] Microsoft. *Tutorial: Write your first analyzer and code fix*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/tutorials/how-to-write-csharp-analyzer-code-fix (visited on 2022-05-06).

[63] Codingvila. *C# Code Analysis Using Roslyn Syntax Trivia*. URL: https://www.codingvila.com/2021/04/csharp-code-analysis-using-roslyn-syntax-trivia.html (visited on 2022-04-22).

[64] Jacob Anderson. *CUP*. URL: https://github.com/BeyondOrdinary/CUP (visited on 2022-05-10).

[65] Olivier Duhart et al. *csly*. URL: https://github.com/b3b00/csly (visited on 2022-05-10).

[66] Jakob Nielsen. *Why You Only Need to Test with 5 Users*. 2000-03-18. URL: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/ (visited on 2020-11-04).

[67] Svetomir Kurtev, Tommy Aagaard Christensen, and Bent Thomsen. "Discount method for programming language evaluation". In: *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. 2016, pp. 1–8.

[68] Thomas Hamilton. *What is Test Driven Development (TDD)? Tutorial with Example*. 2022-04-30. URL: https://www.guru99.com/test-driven-development.html (visited on 2022-05-30).

[69] Gregor Kiczales. "Aspect-oriented programming". In: *ACM Computing Surveys (CSUR)* 28.4es (1996), 154–es.

Appendix A

Appendix

A.1 P-values for Microbenchmarks

A.1.1 Lambda Expressions Outside Loop

Elapsed Time *p*-Values	Lambda	Lambda Closure	Lambda Action	Lambda Parameter	Lambda Delegate
Lambda	-	<0,05	<0,05	<0,05	<0,05
Lambda Closure	<0,05	-	0,877	0,724	<0,05
Lambda Action	<0,05	0,877	-	0,831	<0,05
Lambda Parameter	<0,05	0,724	0,831	-	<0,05
Lambda Delegate	<0,05	<0,05	<0,05	<0,05	-

Table A.1: Table showing the *p*-values for the group Lambda Expression with regards to Elapsed Time.

Package Energy *p*-Values	Lambda	Lambda Closure	Lambda Action	Lambda Parameter	Lambda Delegate
Lambda	-	<0,05	<0,05	<0,05	<0,05
Lambda Closure	<0,05	-	<0,05	<0,05	<0,05
Lambda Action	<0,05	<0,05	-	<0,05	<0,05
Lambda Parameter	<0,05	<0,05	<0,05	-	<0,05
Lambda Delegate	<0,05	<0,05	<0,05	<0,05	-

Table A.2: Table showing the *p*-values for the group Lambda Expression with regards to Package Energy.

DRAM Energy *p*-Values	Lambda	Lambda Closure	Lambda Action	Lambda Parameter	Lambda Delegate
Lambda	-	<0,05	<0,05	<0,05	<0,05
Lambda Closure	<0,05	-	0,781	0,430	<0,05
Lambda Action	<0,05	0,781	-	0,468	<0,05
Lambda Parameter	<0,05	0,430	0,468	-	<0,05
Lambda Delegate	<0,05	<0,05	<0,05	<0,05	-

Table A.3: Table showing the *p*-values for the group Lambda Expression with regards to DRAM Energy.

A.1.2 Lambda Expressions Inside Loop

Elapsed Time *p*-Values	Inside Loop Lambda	Inside Loop Lambda Closure	Inside Loop Lambda Action	Inside Loop Lambda Parameter	Inside Loop Lambda Delegate
Inside Loop Lambda	-	<0,05	<0,05	<0,05	<0,05
Inside Loop Lambda Closure	<0,05	-	<0,05	<0,05	<0,05
Inside Loop Lambda Action	<0,05	<0,05	-	<0,05	<0,05
Inside Loop Lambda Parameter	<0,05	<0,05	<0,05	-	<0,05
Inside Loop Lambda Delegate	<0,05	<0,05	<0,05	<0,05	-

Table A.4: Table showing the *p*-values for the group Lambda Expression Inside Loop with regards to Elapsed Time.

Package Energy *p*-Values	Inside Loop Lambda	Inside Loop Lambda Closure	Inside Loop Lambda Action	Inside Loop Lambda Parameter	Inside Loop Lambda Delegate
Inside Loop Lambda	-	<0,05	<0,05	0,355	<0,05
Inside Loop Lambda Closure	<0,05	-	<0,05	<0,05	<0,05
Inside Loop Lambda Action	<0,05	<0,05	-	<0,05	<0,05
Inside Loop Lambda Parameter	0,355	<0,05	<0,05	-	<0,05
Inside Loop Lambda Delegate	<0,05	<0,05	<0,05	<0,05	-

Table A.5: Table showing the *p*-values for the group Lambda Expression Inside Loop with regards to Package Energy.

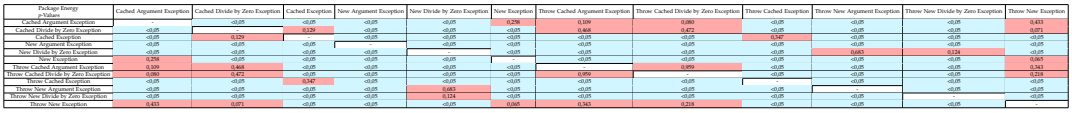
DRAM Energy *p*-Values	Inside Loop Lambda	Inside Loop Lambda Closure	Inside Loop Lambda Action	Inside Loop Lambda Parameter	Inside Loop Lambda Delegate
Inside Loop Lambda	-	<0,05	<0,05	<0,05	<0,05
Inside Loop Lambda Closure	<0,05	-	0,330	<0,05	<0,05
Inside Loop Lambda Action	<0,05	0,330	-	<0,05	<0,05
Inside Loop Lambda Parameter	<0,05	<0,05	<0,05	-	<0,05
Inside Loop Lambda Delegate	<0,05	<0,05	<0,05	<0,05	-

Table A.6: Table showing the *p*-values for the group Lambda Expression Inside Loop with regards to DRAM Energy.

A.1.3 Throwing and Catching Exceptions

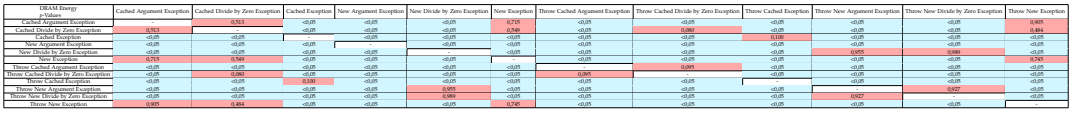
Elapsed Time *p*-Values	Cached Argument Exception	Cached Divide by Zero Exception	Cached Exception	New Argument Exception	New Divide by Zero Exception	New Exception	Throw Cached Argument Exception	Throw Cached Divide by Zero Exception	Throw Cached Exception	Throw New Argument Exception	Throw New Divide by Zero Exception	Throw New Exception
Cached Argument Exception		0,875	<0,05	<0,05	<0,05	0,530	<0,05	0,154	<0,05	<0,05	<0,05	0,412
Cached Divide by Zero Exception	0,875	-	<0,05	<0,05	<0,05	0,292	<0,05	0,123	<0,05	<0,05	<0,05	0,239
Cached Exception	<0,05	<0,05		<0,05	<0,05	<0,05	<0,05	<0,05	0,190	<0,05	<0,05	<0,05
New Argument Exception	<0,05	<0,05	<0,05		0,166	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
New Divide by Zero Exception	<0,05	<0,05	<0,05	0,166		<0,05	<0,05	<0,05	<0,05	0,296	0,236	<0,05
New Exception	0,530	0,292	<0,05	<0,05	<0,05		<0,05	<0,05	<0,05	<0,05	<0,05	0,463
Throw Cached Argument Exception	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05		0,357	<0,05	<0,05	<0,05	<0,05
Throw Cached Divide by Zero Exception	0,154	0,123	<0,05	<0,05	<0,05	<0,05	0,357		<0,05	<0,05	<0,05	<0,05
Throw Cached Exception	<0,05	<0,05	0,190	<0,05	<0,05	<0,05	<0,05	<0,05		<0,05	<0,05	<0,05
Throw New Argument Exception	<0,05	<0,05	<0,05	<0,05	0,296	<0,05	<0,05	<0,05	<0,05		0,940	<0,05
Throw New Divide by Zero Exception	<0,05	<0,05	<0,05	<0,05	0,236	<0,05	<0,05	<0,05	<0,05	0,940		<0,05
Throw New Exception	0,412	0,239	<0,05	<0,05	<0,05	0,463	<0,05	<0,05	<0,05	<0,05	<0,05	

Table A.7: Table showing the *p*-values for the group Exception with regards to Elapsed Time.



Package Energy *p*-Values	Cached Argument Exception	Cached Divide by Zero Exception	Cached Exception	New Argument Exception	New Divide by Zero Exception	New Exception	Throw Cached Argument Exception	Throw Cached Divide by Zero Exception	Throw Cached Exception	Throw New Argument Exception	Throw New Divide by Zero Exception	Throw New Exception
Cached Argument Exception	-	<0,05	<0,05	<0,05	<0,05	0,258	0,109	0,080	<0,05	<0,05	<0,05	0,433
Cached Divide by Zero Exception	<0,05	-	0,129	<0,05	<0,05	<0,05	0,468	0,472	<0,05	<0,05	<0,05	0,071
Cached Exception	<0,05	0,129	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,347	<0,05	<0,05	<0,05
New Argument Exception	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
New Divide by Zero Exception	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	0,683	0,124	<0,05
New Exception	0,258	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,065
Throw Cached Argument Exception	0,109	0,468	<0,05	<0,05	<0,05	<0,05	-	0,959	<0,05	<0,05	<0,05	0,343
Throw Cached Divide by Zero Exception	0,080	0,472	<0,05	<0,05	<0,05	<0,05	0,959	-	<0,05	<0,05	<0,05	0,218
Throw Cached Exception	<0,05	<0,05	0,347	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Throw New Argument Exception	<0,05	<0,05	<0,05	<0,05	0,683	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Throw New Divide by Zero Exception	<0,05	<0,05	<0,05	<0,05	0,124	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Throw New Exception	0,433	0,071	<0,05	<0,05	<0,05	0,065	0,343	0,218	<0,05	<0,05	<0,05	-

Table A.8: Table showing the *p*-values for the group Exception with regards to Package Energy.



DRAM Energy *p*-Values	Cached Argument Exception	Cached Divide by Zero Exception	Cached Exception	New Argument Exception	New Divide by Zero Exception	New Exception	Throw Cached Argument Exception	Throw Cached Divide by Zero Exception	Throw Cached Exception	Throw New Argument Exception	Throw New Divide by Zero Exception	Throw New Exception
Cached Argument Exception	-	0,513	<0,05	<0,05	<0,05	0,715	<0,05	<0,05	<0,05	<0,05	<0,05	0,935
Cached Divide by Zero Exception	0,513	-	<0,05	<0,05	<0,05	0,549	<0,05	0,080	<0,05	<0,05	<0,05	0,464
Cached Exception	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,100	<0,05	<0,05	<0,05
New Argument Exception	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
New Divide by Zero Exception	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	0,955	0,989	<0,05
New Exception	0,715	0,549	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,745
Throw Cached Argument Exception	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,095	<0,05	<0,05	<0,05	<0,05
Throw Cached Divide by Zero Exception	<0,05	0,080	<0,05	<0,05	<0,05	<0,05	0,095	-	<0,05	<0,05	<0,05	<0,05
Throw Cached Exception	<0,05	<0,05	0,100	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Throw New Argument Exception	<0,05	<0,05	<0,05	<0,05	0,955	<0,05	<0,05	<0,05	<0,05	-	0,927	<0,05
Throw New Divide by Zero Exception	<0,05	<0,05	<0,05	<0,05	0,989	<0,05	<0,05	<0,05	<0,05	0,927	-	<0,05
Throw New Exception	0,935	0,464	<0,05	<0,05	<0,05	0,745	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.9: Table showing the *p*-values for the group Exception with regards to DRAM Energy.

A.1.4 Exception Creation

Elapsed Time *p*-Values	Create Argument Exception	Create Divide by Zero Exception	Create Exception
Create Argument Exception	-	<0,05	<0,05
Create Divide by Zero Exception	<0,05	-	<0,05
Create Exception	<0,05	<0,05	-

Table A.10: Table showing the *p*-values for the group Exception Creation with regards to Elapsed Time.

Package Energy *p*-Values	Create Argument Exception	Create Divide by Zero Exception	Create Exception
Create Argument Exception	-	<0,05	<0,05
Create Divide by Zero Exception	<0,05	-	<0,05
Create Exception	<0,05	<0,05	-

Table A.11: Table showing the *p*-values for the group Exception Creation with regards to Package Energy.

DRAM Energy *p*-Values	Create Argument Exception	Create Divide by Zero Exception	Create Exception
Create Argument Exception	-	<0,05	<0,05
Create Divide by Zero Exception	<0,05	-	<0,05
Create Exception	<0,05	<0,05	-

Table A.12: Table showing the *p*-values for the group Exception Creation with regards to DRAM Energy.

A.2 Larger Benchmark Changes

The changes made to the larger benchmarks have been summarized to the following list.

- `uint` variables should replace other integer variables when possible.
- If `uint` is not possible, `ulong` should be used.
- If unsigned integers are not possible, `int`, `nint` or `long` should be used.
- `double` should replace other types of decimal types.
- parameters should replace other types of variables when possible.
- If parameters can not be used local variables should replace `instance` and `static` variables when possible.
- If parameters and local variables can not be used, `static` variables should be used when possible.
- `if` statements should be replaced with `switch` statements when possible.
- `foreach` loops should be used when elements in a collection is gone through.
- `LINQ` should be replaced when possible.
- `array` should replace any type of `list` if possible.
- If `array` is not possible, `List` should replace any type of `list` if possible.
- `HashSet` should replace any type of `set` when possible.
- `Dictionary` should replace any type of `table` when possible.
- Replace occurences of `string` concatenation with `StringBuilder` or `string` interpolation.
- Replace boxed integer datatypes with unboxed integer datatypes when possible.

- If replacing boxed datatypes with unboxed datatypes is not possible, use boxed `uint` if possible.
- If using boxed `uint` is not possible, use boxed `ulong` if possible.
- If using boxed `ulong` is not possible, use boxed `int` or `long` if possible.
- Replace all boxed floating point datatypes with the unboxed `double` datatype if possible.
- Replace boxed boolean datatypes with unboxed boolean datatypes if possible.
- Replace `reflection` with other types of invocation when possible.
- If replacing `reflection` is not possible, wrap `reflection` in `delegates` if possible.
- Replace `classes` and `records` with `structs`, if creating `objects` and invoking `methods` is the most prevalent.
- Replace `structs` with `classes` if accessing fields is the most prevalent.
- Remove `try-catch` blocks if possible.
- Replace `exceptions` with `if` statements when possible.
- Avoid using variables outside the `lambda expression` when possible.
- Replace `ArgumentException` and `DivideByZeroException` with `Exception` when possible.
- Reuse `exceptions` if thrown more than once.

A.2.1 Batches

The different batches of changes are divided into, and each of these batches is tested individually to see the impact each of them has.

Datatypes

- `uint` variables should replace other integer variables when possible.
- If `uint` is not possible, `ulong` should be used.
- If unsigned integers are not possible, `int`, `nint` or `long` should be used.
- `double` should replace other types of decimal types.
- parameters should replace other types of variables when possible.
- If parameters can not be used local variables should replace `instance` and `static` variables when possible.
- If parameters and local variables can not be used, `static` variables should be used when possible.
- Replace occurences of `string` concatenation with `StringBuilder` or `string` interpolation.
- Replace boxed integer datatypes with unboxed integer datatypes when possible.
- If replacing boxed datatypes with unboxed datatypes is not possible, use boxed `uint` if possible.
- If using boxed `uint` is not possible, use boxed `ulong` if possible.
- If using boxed `ulong` is not possible, use boxed `int` or `long` if possible.
- Replace all boxed floating point datatypes with the unboxed `double` datatype if possible.
- Replace boxed boolean datatypes with unboxed boolean datatypes if possible.

Selection

- `if` statements should be replaced with `switch` statements when possible.

Loops

- `foreach` loops should be used when elements in a collection is gone through.

LINQ

- `LINQ` should be replaced when possible.

Collections

- `array` should replace any type of `list` if possible.
- If `array` is not possible, `List` should replace any type of `list` if possible.
- `HashSet` should replace any type of `set` when possible.
- `Dictionary` should replace any type of `table` when possible.

Invocation

- Replace `reflection` with other types of invocation when possible.
- If replacing `reflection` is not possible, wrap `reflection` in `delegates` if possible.
- Avoid using variables outside the `lambda expression` when possible.

Objects

- Replace `classes` and `records` with `structs`, if creating `objects` and invoking `methods` is the most prevalent.
- Replace `structs` with `classes` if accessing fields is the most prevalent.

Exceptions

- Remove `try-catch` blocks if possible.
- Replace `exceptions` with `if` statements when possible.
- Replace `ArgumentException` and `DivideByZeroException` with `Exception` when possible.

- Reuse `exceptions` if thrown more than once.

A.3 P-values for the larger benchmarks

A.3.1 2048

Elapsed Time *p*-Values	All 2048	Data Type 2048	Default 2048	Foreach 2048	Object 2048	Switch 2048
All 2048	-	0,512	<0,05	<0,05	0,160	<0,05
Data Type 2048	0,512	-	<0,05	0,492	0,053	<0,05
Default 2048	<0,05	<0,05	-	0,152	0,801	0,097
Foreach 2048	<0,05	0,492	0,152	-	0,387	<0,05
Object 2048	0,160	0,053	0,801	0,387	-	0,136
Switch 2048	<0,05	<0,05	0,097	<0,05	0,136	-

Table A.13: Table showing the *p*-values for the group 2048 with regards to Elapsed Time.

Package Energy *p*-Values	All 2048	Data Type 2048	Default 2048	Foreach 2048	Object 2048	Switch 2048
All 2048	-	0,472	0,645	0,782	0,155	<0,05
Data Type 2048	0,472	-	0,179	0,313	<0,05	<0,05
Default 2048	0,645	0,179	-	0,838	0,127	<0,05
Foreach 2048	0,782	0,313	0,838	-	0,161	<0,05
Object 2048	0,155	<0,05	0,127	0,161	-	0,176
Switch 2048	<0,05	<0,05	<0,05	<0,05	0,176	-

Table A.14: Table showing the *p*-values for the group 2048 with regards to Package Energy.

DRAM Energy *p*-Values	All 2048	Data Type 2048	Default 2048	Foreach 2048	Object 2048	Switch 2048
All 2048	-	0,363	<0,05	<0,05	0,095	<0,05
Data Type 2048	0,363	-	<0,05	0,549	<0,05	<0,05
Default 2048	<0,05	<0,05	-	0,125	0,676	0,077
Foreach 2048	<0,05	0,549	0,125	-	0,270	<0,05
Object 2048	0,095	<0,05	0,676	0,270	-	0,180
Switch 2048	<0,05	<0,05	0,077	<0,05	0,180	-

Table A.15: Table showing the *p*-values for the group 2048 with regards to DRAM Energy.

A.3.2 21

Elapsed Time *p*-Values	All 21	Datatype 21	Default 21	Exception 21	Switch 21
All 21	-	0,497	<0,05	<0,05	<0,05
Datatype 21	0,497	-	<0,05	<0,05	<0,05
Default 21	<0,05	<0,05	-	0,317	<0,05
Exception 21	<0,05	<0,05	0,317	-	<0,05
Switch 21	<0,05	<0,05	<0,05	<0,05	-

Table A.16: Table showing the *p*-values for the group 21 with regards to Elapsed Time.

Package Energy *p*-Values	All 21	Datatype 21	Default 21	Exception 21	Switch 21
All 21	-	<0,05	<0,05	<0,05	<0,05
Datatype 21	<0,05	-	<0,05	<0,05	<0,05
Default 21	<0,05	<0,05	-	<0,05	0,689
Exception 21	<0,05	<0,05	<0,05	-	0,489
Switch 21	<0,05	<0,05	0,689	0,489	-

Table A.17: Table showing the *p*-values for the group 21 with regards to Package Energy.

DRAM Energy *p*-Values	All 21	Datatype 21	Default 21	Exception 21	Switch 21
All 21	-	0,744	<0,05	<0,05	<0,05
Datatype 21	0,744	-	<0,05	<0,05	<0,05
Default 21	<0,05	<0,05	-	0,139	<0,05
Exception 21	<0,05	<0,05	0,139	-	<0,05
Switch 21	<0,05	<0,05	<0,05	<0,05	-

Table A.18: Table showing the *p*-values for the group 21 with regards to DRAM Energy.

A.3.3 4-Rings or 4-Squares Puzzle

Elapsed Time *p*-Values	All Four Squares	Data Type Four Squares	Default Four Squares	LINQ Four Squares
All Four Squares	-	<0,05	<0,05	<0,05
Data Type Four Squares	<0,05	-	<0,05	<0,05
Default Four Squares	<0,05	<0,05	-	<0,05
LINQ Four Squares	<0,05	<0,05	<0,05	-

Table A.19: Table showing the *p*-values for the group Four Squares with regards to Elapsed Time.

Package Energy *p*-Values	All Four Squares	Data Type Four Squares	Default Four Squares	LINQ Four Squares
All Four Squares	-	<0,05	<0,05	<0,05
Data Type Four Squares	<0,05	-	<0,05	<0,05
Default Four Squares	<0,05	<0,05	-	<0,05
LINQ Four Squares	<0,05	<0,05	<0,05	-

Table A.20: Table showing the *p*-values for the group Four Squares with regards to Package Energy.

DRAM Energy *p*-Values	All Four Squares	Data Type Four Squares	Default Four Squares	LINQ Four Squares
All Four Squares	-	<0,05	<0,05	0,103
Data Type Four Squares	<0,05	-	<0,05	<0,05
Default Four Squares	<0,05	<0,05	-	<0,05
LINQ Four Squares	0,103	<0,05	<0,05	-

Table A.21: Table showing the *p*-values for the group Four Squares with regards to DRAM Energy.

A.3.4 99 Bottles of Beer

Elapsed Time *p*-Values	All 99 Bottles of Beer	Data Type 99 Bottles of Beer	Default 99 Bottles of Beer	Invocation 99 Bottles of Beer
All 99 Bottles of Beer	-	<0,05	<0,05	<0,05
Data Type 99 Bottles of Beer	<0,05	-	<0,05	<0,05
Default 99 Bottles of Beer	<0,05	<0,05	-	<0,05
Invocation 99 Bottles of Beer	<0,05	<0,05	<0,05	-

Table A.22: Table showing the *p*-values for the group 99 Bottles of Beer with regards to Elapsed Time.

Package Energy *p*-Values	All 99 Bottles of Beer	Data Type 99 Bottles of Beer	Default 99 Bottles of Beer	Invocation 99 Bottles of Beer
All 99 Bottles of Beer	-	<0,05	<0,05	<0,05
Data Type 99 Bottles of Beer	<0,05	-	<0,05	<0,05
Default 99 Bottles of Beer	<0,05	<0,05	-	0,974
Invocation 99 Bottles of Beer	<0,05	<0,05	0,974	-

Table A.23: Table showing the *p*-values for the group 99 Bottles of Beer with regards to Package Energy.

DRAM Energy *p*-Values	All 99 Bottles of Beer	Data Type 99 Bottles of Beer	Default 99 Bottles of Beer	Invocation 99 Bottles of Beer
All 99 Bottles of Beer	-	<0,05	<0,05	<0,05
Data Type 99 Bottles of Beer	<0,05	-	<0,05	<0,05
Default 99 Bottles of Beer	<0,05	<0,05	-	<0,05
Invocation 99 Bottles of Beer	<0,05	<0,05	<0,05	-

Table A.24: Table showing the *p*-values for the group 99 Bottles of Beer with regards to DRAM Energy.

A.3.5 Determine if a String has All the Same Characters

Elapsed Time *p*-Values	All Equal Strings	Data Type Equal Strings	Default Equal Strings	For Each Equal Strings
All Equal Strings	-	<0,05	<0,05	<0,05
Data Type Equal Strings	<0,05	-	<0,05	<0,05
Default Equal Strings	<0,05	<0,05	-	<0,05
For Each Equal Strings	<0,05	<0,05	<0,05	-

Table A.25: Table showing the *p*-values for the group Equal Strings with regards to Elapsed Time.

Package Energy *p*-Values	All Equal Strings	Data Type Equal Strings	Default Equal Strings	For Each Equal Strings
All Equal Strings	-	<0,05	<0,05	<0,05
Data Type Equal Strings	<0,05	-	<0,05	<0,05
Default Equal Strings	<0,05	<0,05	-	<0,05
For Each Equal Strings	<0,05	<0,05	<0,05	-

Table A.26: Table showing the *p*-values for the group Equal Strings with regards to Package Energy.

DRAM Energy *p*-Values	All Equal Strings	Data Type Equal Strings	Default Equal Strings	For Each Equal Strings
All Equal Strings	-	<0,05	<0,05	<0,05
Data Type Equal Strings	<0,05	-	<0,05	<0,05
Default Equal Strings	<0,05	<0,05	-	<0,05
For Each Equal Strings	<0,05	<0,05	<0,05	-

Table A.27: Table showing the *p*-values for the group Equal Strings with regards to DRAM Energy.

A.3.6 Dijkstra's Algorithm

Elapsed Time *p*-Values	All Dijkstra	Collections Dijkstra	Data Type Dijkstra	Default Dijkstra	LINQ Dijkstra	Objects Dijkstra
All Dijkstra	-	<0,05	<0,05	<0,05	<0,05	<0,05
Collections Dijkstra	<0,05	-	<0,05	<0,05	<0,05	<0,05
Data Type Dijkstra	<0,05	<0,05	-	<0,05	<0,05	<0,05
Default Dijkstra	<0,05	<0,05	<0,05	-	<0,05	<0,05
LINQ Dijkstra	<0,05	<0,05	<0,05	<0,05	-	<0,05
Objects Dijkstra	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.28: Table showing the *p*-values for the group Dijkstra with regards to Elapsed Time.

Package Energy *p*-Values	All Dijkstra	Collections Dijkstra	Data Type Dijkstra	Default Dijkstra	LINQ Dijkstra	Objects Dijkstra
All Dijkstra	-	<0,05	<0,05	<0,05	<0,05	<0,05
Collections Dijkstra	<0,05	-	<0,05	<0,05	<0,05	<0,05
Data Type Dijkstra	<0,05	<0,05	-	<0,05	<0,05	<0,05
Default Dijkstra	<0,05	<0,05	<0,05	-	<0,05	<0,05
LINQ Dijkstra	<0,05	<0,05	<0,05	<0,05	-	<0,05
Objects Dijkstra	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.29: Table showing the *p*-values for the group Dijkstra with regards to Package Energy.

DRAM Energy *p*-Values	All Dijkstra	Collections Dijkstra	Data Type Dijkstra	Default Dijkstra	LINQ Dijkstra	Objects Dijkstra
All Dijkstra	-	<0,05	<0,05	<0,05	<0,05	<0,05
Collections Dijkstra	<0,05	-	<0,05	<0,05	<0,05	<0,05
Data Type Dijkstra	<0,05	<0,05	-	<0,05	<0,05	<0,05
Default Dijkstra	<0,05	<0,05	<0,05	-	<0,05	<0,05
LINQ Dijkstra	<0,05	<0,05	<0,05	<0,05	-	<0,05
Objects Dijkstra	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.30: Table showing the *p*-values for the group Dijkstra with regards to DRAM Energy.

A.3.7 Happy Numbers

Elapsed Time *p*-Values	All Happy Number	Collection Happy Number	Data Type Happy Number	Default Happy Number
All Happy Number	-	<0,05	<0,05	<0,05
Collection Happy Number	<0,05	-	<0,05	<0,05
Data Type Happy Number	<0,05	<0,05	-	<0,05
Default Happy Number	<0,05	<0,05	<0,05	-

Table A.31: Table showing the *p*-values for the group Happy Numbers with regards to Elapsed Time.

Package Energy *p*-Values	All Happy Number	Collection Happy Number	Data Type Happy Number	Default Happy Number
All Happy Number	-	<0,05	<0,05	<0,05
Collection Happy Number	<0,05	-	<0,05	<0,05
Data Type Happy Number	<0,05	<0,05	-	<0,05
Default Happy Number	<0,05	<0,05	<0,05	-

Table A.32: Table showing the *p*-values for the group Happy Numbers with regards to Package Energy.

DRAM Energy *p*-Values	All Happy Number	Collection Happy Number	Data Type Happy Number	Default Happy Number
All Happy Number	-	<0,05	<0,05	<0,05
Collection Happy Number	<0,05	-	<0,05	<0,05
Data Type Happy Number	<0,05	<0,05	-	<0,05
Default Happy Number	<0,05	<0,05	<0,05	-

Table A.33: Table showing the *p*-values for the group Happy Numbers with regards to DRAM Energy.

A.3.8 Introspection

Elapsed Time *p*-Values	All Introspection	Data Type Introspection	Default Introspection	Invocation Introspection
All Introspection	-	0,134	<0,05	0,060
Data Type Introspection	0,134	-	0,360	0,463
Default Introspection	<0,05	0,360	-	0,746
Invocation Introspection	0,060	0,463	0,746	-

Table A.34: Table showing the *p*-values for the group Introspection with regards to Elapsed Time.

Package Energy *p*-Values	All Introspection	Data Type Introspection	Default Introspection	Invocation Introspection
All Introspection	-	<0,05	0,836	0,063
Data Type Introspection	<0,05	-	<0,05	0,220
Default Introspection	0,836	<0,05	-	<0,05
Invocation Introspection	0,063	0,220	<0,05	-

Table A.35: Table showing the *p*-values for the group Introspection with regards to Package Energy.

DRAM Energy *p*-Values	All Introspection	Data Type Introspection	Default Introspection	Invocation Introspection
All Introspection	-	0,140	0,058	<0,05
Data Type Introspection	0,140	-	0,441	0,232
Default Introspection	0,058	0,441	-	0,895
Invocation Introspection	<0,05	0,232	0,895	-

Table A.36: Table showing the *p*-values for the group Introspection with regards to DRAM Energy.

A.3.9 World Cup Group Stage

Elapsed Time *p*-Values	All World Cup Stage	Data Type World Cup Stage	Default World Cup Stage	LINQ World Cup Stage
All World Cup Stage	-	<0,05	<0,05	<0,05
Data Type World Cup Stage	<0,05	-	<0,05	<0,05
Default World Cup Stage	<0,05	<0,05	-	<0,05
LINQ World Cup Stage	<0,05	<0,05	<0,05	-

Table A.37: Table showing the *p*-values for the group World Cup Group Stage with regards to Elapsed Time.

Package Energy *p*-Values	All World Cup Stage	Data Type World Cup Stage	Default World Cup Stage	LINQ World Cup Stage
All World Cup Stage	-	<0,05	<0,05	<0,05
Data Type World Cup Stage	<0,05	-	<0,05	<0,05
Default World Cup Stage	<0,05	<0,05	-	<0,05
LINQ World Cup Stage	<0,05	<0,05	<0,05	-

Table A.38: Table showing the *p*-values for the group World Cup Group Stage with regards to Package Energy.

DRAM Energy *p*-Values	All World Cup Stage	Data Type World Cup Stage	Default World Cup Stage	LINQ World Cup Stage
All World Cup Stage	-	<0,05	<0,05	<0,05
Data Type World Cup Stage	<0,05	-	<0,05	<0,05
Default World Cup Stage	<0,05	<0,05	-	<0,05
LINQ World Cup Stage	<0,05	<0,05	<0,05	-

Table A.39: Table showing the *p*-values for the group World Cup Group Stage with regards to DRAM Energy.

A.3.10 Overview of Changes to the larger benchmarks

In this section, we create an overview of the expected effects of each change that is made in the large benchmarks. These do not include all of the individual changes from the batches from Section A.2.1, as we have found that a lot of the changes from the batches are not used in the benchmarks from Rosetta Code [39].

Batch	Change	Effect in Microbenchmark
Datatypes	Using uint instead of int.	23%, from 33.452μJ to 25.862μJ.
	Using StringBuilder instead of Concatenation Operator (+).	36%, from 1.012.054μJ to 649.596μJ.

	Using StringBuilder instead of Interpolation.	31%, from 937.184μJ to 649.596μJ.
	Using StringBuilder instead of String.Format.	74%, from 2.452.011μJ to 649.596μJ.
	Using StringBuilder instead of String.Concat.	57%, from 1.498.696μJ to 649.596μJ.
Selection	Using switch instead of if.	41%, from 31.751μJ to 18.620μJ.
Loops	Using foreach instead of for.	18%, from 64.485μJ to 53.031μJ.
LINQ	Removing and replacing LINQ.	90%, from 88.412.171μJ to 9.658.162μJ.
Collections	Change List to array	16%, from 10.531.627μJ to 8.794.373μJ.
Invocation	Using parameters in lambda expressions instead of closure.	13%, from 200.548μJ to 173.911μJ.
	Wrapping reflection in delegate.	98%, from 1.506.979μJ to 29.323μJ.
Objects	Using struct instead of class.	50% when accessing methods, from 18.222μJ to 9.201μJ. 90% when creating instances, from 64.624μJ to 6.749μJ. -35% when accessing fields, from 4.878μJ to 6.617μJ.
Exceptions	Using if instead of try-catch.	99,8%, from 28.131.047μJ to 64.449μJ.
	Removing try-catch block.	53%, from 66.096μJ to 31.295μJ.

Table A.40: The changes in the larger benchmarks and their respective effect in the microbenchmarks.

A.4 Formal Suggestions for Energy Analyzer

In this section, the formal suggestions for *Energy Analyzer* is presented. This is used for *Energy Analyzer* in Chapter 5.

The original suggestions from Section A.2 are used as a starting point

and changed based on the findings found in Section 4.2.

This means these suggestions are based on the microbenchmarks when no larger benchmarks have used the suggestion, and on larger benchmarks when the suggestion has been used in one of those.

Datatypes

- `uint` variables should replace other integer variables when possible, except in `Lists` where `int` should be used.
- If `uint` is not possible, `ulong` should be used.
- If unsigned integers are not possible, `int`, `nint` or `long` should be used.
- `double` should replace other types of decimal types.
- Replace occurences of `string` concatenation with `StringBuilder` when three or more `strings` are concatenated.
- If two `strings` are concatenated, `string` interpolation should be used.
- If a new `StringBuilder` is created often enough to trigger garbage collection, `string` interpolation should be used.
- Replace boxed integer datatypes with unboxed integer datatypes when possible.
- If replacing boxed datatypes with unboxed datatypes is not possible, use boxed `uint` if possible.
- If using boxed `uint` is not possible, use boxed `ulong` if possible.
- If using boxed `ulong` is not possible, use boxed `int` or `long` if possible.
- Replace all boxed floating point datatypes with the unboxed `double` datatype if possible.
- Replace boxed boolean datatypes with unboxed boolean datatypes if possible.
- parameters should replace other types of variables when possible.
- If parameters can not be used local variables should replace `instance` and `static` variables when possible.

- If parameters and local variables can not be used, `static` variables should be used when possible.

Selection

- `if` statements should be replaced with `switch` statements when possible.

Loops

- `foreach` loops should be used when elements in a collection is gone through, unless the index is needed, in which case a `for` loop should be used.

LINQ

- `LINQ` should be replaced when possible.

Collections

- `array` should replace any type of `list` if possible.
- If `array` is not possible, `List` should replace any type of `list` if possible.
- `HashSet` should replace any type of `set` when possible.
- `Dictionary` should replace any type of `table` when possible.

Invocation

- Replace `reflection` with other types of invocation when possible.
- If replacing `reflection` is not possible, wrap `reflection` in `delegates` if possible.
- Avoid using variables outside the `lambda expression` when possible.

Objects

- Replace `classes` and `records` with `structs`, if creating `objects` and invoking `methods` is the most prevalent.
- Replace `structs` with `classes` if accessing fields is the most prevalent.

Exceptions

- Remove `try-catch` blocks if possible.
- Replace `exceptions` with `if` statements when possible.
- Replace `ArgumentException` and `DivideByZeroException` with `Exception` when possible.
- Reuse `exceptions` if thrown more than once.

A.5 P-values for Evaluation

A.5.1 CUP

Elapsed Time *p*-Values	Original	All	Code Fixes
Original	-	<0,05	<0,05
All	<0,05	-	<0,05
Code Fixes	<0,05	<0,05	-

Table A.41: Table showing the *p*-values for the group CUP with regards to Elapsed Time.

Package Energy *p*-Values	Original	All	Code Fixes
Original	-	<0,05	<0,05
All	<0,05	-	0,419
Code Fixes	<0,05	0,419	-

Table A.42: Table showing the *p*-values for the group CUP with regards to Package Energy.

DRAM Energy *p*-Values	Original	All	Code Fixes
Original	-	<0,05	<0,05
All	<0,05	-	0,306
Code Fixes	<0,05	0,306	-

Table A.43: Table showing the *p*-values for the group CUP with regards to DRAM Energy.

A.5.2 Sly

Elapsed Time *p*-Values	All	Code Fixes	Original
All	-	<0,05	<0,05
Code Fixes	<0,05	-	0,123
Original	<0,05	0,123	-

Table A.44: Table showing the *p*-values for the group Sly with regards to Elapsed Time.

Package Energy *p*-Values	All	Code Fixes	Original
All	-	0,460	0,101
Code Fixes	0,460	-	0,329
Original	0,101	0,329	-

Table A.45: Table showing the *p*-values for the group Sly with regards to Package Energy.

DRAM Energy *p*-Values	All	Code Fixes	Original
All	-	0,207	0,462
Code Fixes	0,207	-	0,678
Original	0,462	0,678	-

Table A.46: Table showing the *p*-values for the group Sly with regards to DRAM Energy.

A.6 Usability Test Tasks

In this section, the introduction to the usability tests as well as the usability test tasks are presented.

Introduction

Today you will be testing a tool for optimizing the energy usage of C# programs. Namely an analyzer named *EnergyAnalyzer*. You will be handed a list of tasks that we expect you to try to solve on your own while thinking out loud. After you have solved all the tasks we will conduct a short interview regarding other thoughts, improvements, and debriefing. The moderator will be able to help you if you get stuck but otherwise will only observe. By participating in this test you accept that we can use the collected data in our university project. The data will be analyzed anonymously. Thank you so much for your attention and participation in advance.

- Task 1
  - Get an overview of the sample program.
- Task 2
  - Install the package using NuGet.
- Task 3
  - Count and say how many suggestions you can see.
- Task 4

  - Apply all code fixes.

- Task 5
  - Observe and explain how you would use the information provided to solve the remaining suggestions.