Analyzing C# Energy Efficiency of Concurrency and Language Construct Combinations

Specialization in Software

Project Report
cs-22-pt-10-05

Aalborg University
Software

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

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Analyzing C# Energy Efficiency of Concurrency and Language Construct Combinations

Theme:
Energy Aware Programming

Project Period:
Spring Semester 2022

Project Group:
cs-22-pt-10-05

Participant(s):
Aleksander Øster Nielsen
Kasper Jepsen
Rasmus Smit Lindholt

Supervisor(s):
Bent Thomsen, Lone Leth Thomsen

Copies: 1

Page Numbers: 243

Date of Completion:
June 8, 2022

Abstract:

This project investigates the impact on energy consumption of different C# language constructs, with a focus on concurrency constructs. We create microbenchmarks that look at the effects of four different groups of language constructs. We analyze the results of these microbenchmarks to determine the reasons behind the observed behavior. Using the knowledge gathered from the microbenchmarks, and previous research, we create 68 macrobenchmarks which are used to generate results for how different combinations of language constructs affect energy consumption. These benchmarks allow us to generalize the effects of language constructs in microbenchmarks to macrobenchmarks and how language constructs work in conjunction. Results of the macrobenchmarks are used to determine if and why behavior changes from microbenchmarks to macrobenchmarks, and when multiple language constructs are used in conjunction.

Preface

This project has been created by 3 software students in the tenth semester, at Aalborg University, under the theme *Energy Aware Programming* in the period 1st of February 2022 to 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 in this paper is available in our public GitLab repository[1]

Summary

This project involves researching the energy consumption of different language constructs in C#. The energy measurements are taken on a PC running Linux in order to use Running Average Power Limit (RAPL) readings for measuring energy consumption. We test the validity of the measured results by performing two-sample t-tests, Kolmogorov-Smirnov tests, as well as a sanity check using the CPU's expected static energy usage and the runtime of the benchmarks. The focus of the project is on concurrency constructs, and as such, we create microbenchmarks for testing different concurrency constructs with differing workloads. These benchmarks result in multiple data sets which we analyze to explain the shown behavior.

Based on the results from our microbenchmarks, eight macrobenchmarks from Rosetta [2] are chosen and refactored with combinations of three different code changes which have proven to be among the best choices in regard to energy consumption in a category of language constructs, according to our microbenchmarks and previous research [3].

Finding macrobenchmarks that can use concurrency and has a large enough workload has proven a challenge. Most benchmarks on Rosetta are intended for single-threaded use and the total runtime is only a few milliseconds, and as such, we refactor multiple benchmarks to get a larger workload.

We analyze these macrobenchmarks to determine if the effects are consistent with the results of the microbenchmarks and if the energy improvements can be compounded. The energy improvements are compounded, but in many cases, an improvement constitutes an insignificant part of the benchmark which makes it negligible when measured with the rest of the benchmark. We find that the performance and energy efficiency of our concurrent benchmarks are heavily impacted by the size of the workload, which aligns with the findings from our microbenchmarks. Comparing the benchmarks with small and large benchmarks, we find that a threshold workload must be reached before it is worth changing a sequential program into a concurrent one. Additionally, we find that concurrent benchmarks' runtime and energy consumption are not necessarily correlated. This means that a faster program does not necessarily mean a more energy-efficient program.

Contents

Chapter 1

Introduction

Studies have shown that energy demands for systems in Information and Communications Technology (ICT) are expected to rise [4], potentially reaching up to 21% of global energy consumption in 2030. In [5], it was found that energy consumption from data centers had risen by 6% from 2010 to 2018 and that in 2018 the total consumption was 205 TWh, which at that point was 1% of the global energy consumption. Combining this knowledge we see that continuing this development is an increasing concern with regard to our environment.

Several areas of research have found that there are ways in which the energy consumption of programs can be reduced, which would also positively impact the energy consumption of ICT systems. One way is looking at writing energy-efficient software. Research has shown that several areas can reduce energy consumption, among others; coding style, algorithm choice, parallel computing, choice of language constructs, choice of programming language, and utilized programming paradigm [3, 6, 7, 8, 9, 10, 11].

Research has shown that multicore processors can offer performance improvements over single-core options for certain kinds of parallelized tasks. However, with the use of multiple cores also comes unique challenges because these processors impact the way work is scheduled, memory is allocated, and instructions are executed [12]. Multicore processors are at the heart of modern computing platforms, thus their energy efficiency has an important role in the challenge of improving the energy efficiency of software [13]. In [13] it was found that optimizing an application in regards to performance does not inherently increase the energy efficiency of the application when run on a multicore processor. Based on these observations

we further investigate the behavior of parallel applications and if there are steps that can be taken to improve the way an application is coded in regard to parallelism. In this regard, we look at C# on the basis that it is the 2nd most used OOP language [14], at the time of writing. This choice is made as most research into energy efficient software is performed in the Java language [7, 15, 16].

To do this we explore whether the work created in [3] can be used to take steps toward a better understanding of the options for coding parallel applications in an energy-efficient manner.

Furthermore, we look into the generalization of results from microbenchmarks to macrobenchmarks. This is an area of interest as we have not found existing research that looks at how different language constructs behave in anything but microbenchmarks. As part of this, we also look into how language constructs affect one another, as this is once more an area of research that is largely unexplored. We have found one paper [17], which touches upon a similar area, in regards to how combinations of refactorings affect one another.

1.1 Problem Statement

Based on what is discussed in the introduction, we present a problem statement and accompanying research questions:

How does concurrency and related language constructs affect the energy consumption of code in C#?

1. *What are the energy effects of language constructs used in concurrent programming in C#?*
2. *What are the energy effects of combining language constructs?*
3. *How can the energy effects of language constructs be generalized across applications?*
4. *How can the energy differences between language constructs be explained?*

Our problem statement is motivated by the research conducted in [3] and the lack of information regarding the energy efficiency of language constructs in C# concerning concurrency. [3] show that the energy consumption of language constructs utilized for the same functionality differ. We

explore if this is also true regarding language constructs used in concurrent programming in C#. We examine whether different language constructs affect one another and how generalizable they are across applications. The intent is to establish the energy effects of different combinations of language constructs. This is partially inspired by [17], where it is found that some combinations of refactorings had fewer gains for energy consumption than the individual components. If results differ from the expected outcome, according to microbenchmarks, they are analyzed to determine whether changing the size of the benchmark or combining language constructs affects them.

1.2 Work Process

In regards to our work process, we use a hybrid between a plan-driven and agile process [18]. We call it a hybrid because we utilize a loose plan, which only outlines the project instead of having a rigid plan. This process allows us to adapt to unforeseen challenges in our project. Furthermore, it allows us to more easily reconfigure the overall plan. This choice has also been made on the basis that we are not capable of planning the entire project at the beginning of the work period, as multiple variables that are central to the direction of our project might change. If we found out that there is no further information to be gained from continuing in a particular direction, the level of detail must change. This means that initial plans for that area must change. Based on this reasoning we have a partially agile approach to planning which should only account for the near future, with relatively few deadlines.

Some parts of the project are not iterated over to any major degree. In particular, the benchmarks themselves are not iterated over further than them fulfilling their goal of testing a specific language construct. The language construct's behavior on macrobenchmark-level energy consumption is also not subject to extensive iterations. The contents of this report are subject to iteration as we acquire new knowledge, receive feedback from our supervisors, and review the different parts of the report.

We have elected to utilize daily stand-up meetings, which supports us in keeping track of the different project members' current tasks, as well as progress, and potential issues with a given task.

We support our work process through the use of two tools. The first tool is `GitKraken Boards` [19]. `GitKraken Boards` is a kanban board, which we

use for tracking all tasks in the project and which stage they are currently at, with stages being *To Do*, *Doing*, *Review* and *Done*. We are then able to assign project members to tasks. This helps to keep track of our workload as well as support an even distribution of the workload.

To collaborate on our code we utilize `GitLab`. This allows us to keep an overview of the code and its changes during our project. As part of this, we choose to utilize branches. Any new features or fixes to the code must first have a separate branch, and will only be merged with the main branch if a merge request has been submitted and approved.

1.3 Related work

In this section, we discuss related work concerning this project. An overview of work related to energy-aware programming is presented in the project group's previous work [3], while we do not present this overview, we present an overview of the paper itself and its findings. Our previous work does not look at concurrency and as such, we look at work related to concurrency and the energy efficiency of concurrent software. Lastly, we look into existing research regarding the generalizability of energy effects.

Previous work

This report builds on previous work the project group has conducted in the report *Benchmarking C# for Energy Consumption* [3], which aimed to answer the following problem statement:

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

A framework for measuring the energy consumption and elapsed time of benchmarks was constructed. The framework is capable of calculating whether there is a significant difference between benchmarks within the same group of benchmarks. Energy measurements are retrieved via Intel's Running Average Power Limit (RAPL) interface. The framework was used on 316 benchmarks across 56 subgroups and their results were later analyzed to uncover reasons for differing results. The results in [3] show that there are differences between some language constructs that are used for similar purposes. For instance, using a switch statement instead of an if

statement can be twice as fast, and use only two-thirds of the energy. Using `string.Format` uses about 2,5 times more energy than string interpolation, and using `StringBuilder` is the most energy-efficient way of concatenating strings.

Concurrency

In [16] the energy efficiency of different concurrency constructs in Java is explored, as well as the relation between performance and energy efficiency. Three thread management constructs are explored, namely *explicit threading* where programmers manually map units of work to threads, *thread pooling* where the programmer creates an often fixed-size pool of threads and submits units of work to the pool, and *work stealing* which is similar to a thread pool except all threads maintain their own buffer of units of work, and if that buffer becomes empty, its maintaining thread may steal work from other threads. It is concluded that faster execution time does not mean less energy consumption, as 6 of 9 experiments are the most energy-efficient when using a sequential version. Using parallel execution however leads to improved energy/performance trade-off for most types of workloads, while the size of the data processed also has a non-linear impact on the energy usage. The different thread management constructs have different impacts on energy consumption. For benchmarks bound by I/O, the *explicit threading* approach is the most energy-efficient, whereas *work stealing* is the least energy-efficient, while the results are opposite for benchmarks that benefit greatly by being parallelized.

In [6] both synchronous and concurrent benchmarks are performed, and different thread management constructs in Haskell are used. It is concluded that faster execution time does not always mean less energy usage regarding concurrent benchmarks. There is no overall winner of thread management constructs, as it varies between benchmarks which one is best. In Haskell, capabilities are virtual processors managed by the Haskell runtime, and using more capabilities than the number of CPU cores significantly decreases performance.

In [13] the energy proportionality of multicore systems is explored. It is found that optimizing for performance alone results in a considerable increase in energy usage, and optimizing only for energy usage significantly degrades performance. A methodology for determining the number of Pareto-optimal solutions is created, and a qualitative dynamic energy model which employs performance monitoring counters to discover these

Pareto-optimal solutions is also proposed.

In [20] an overview of performance measurement and analytical modeling techniques for multi-core processors is presented. The performance of multi-core benchmarks is compared to an estimation created using Amdahl's law and Gustafson's law. Through the use of these laws, they calculate an estimation of the speedup caused by using multiple cores. The calculated estimation is compared to the performance of a benchmark application. It is found that the estimations are pessimistic when compared to the observed benchmark. Although the estimations are pessimistic they are usable for performance insights into parallelized applications. In [21] Amdahl's law is extended upon and used to create estimations for energy and power. The difference between three many-core design styles is demonstrated, calculating the estimated performance and energy efficiency of each style and comparing the results. [21] shows that when knowing the percentage of parallelism available in an application prior to execution it is possible to find an optimal number of active cores, thus maximizing the performance for a given cooling capacity and energy in a system.

Generalizability

Another related area of work is the idea of generalizability. As we want to determine how the energy effects of language constructs can be generalized across applications, we look into studies that have looked at similar areas. This helps establish what has already been found in the literature, what to expect, and how to approach our own generalizations of language construct energy effects.

In [22] a study explores how common refactorings impact the energy efficiency of different applications. The research stems from a lacking focus on how maintainability concerning refactorings impacts energy efficiency. This is achieved by measuring different applications' energy consumption before and after refactorings have been applied. It was found that all considered refactorings could impact energy efficiency, however, the impacts were inconsistent. For all but one of the refactorings, the impact could be both positive and negative and the likelihood of an increase or decrease in energy efficiency was deemed to be approximately the same.

In [17] the study looks at different refactorings and their impact on the energy efficiency of Android applications. The refactorings are targeted to compact energy-inefficient patterns. [17] studies the energy efficiency of applications based on the impact of different individual refactorings com-

pared against each other and combinations of these. It was found that refactoring individual energy inefficient patterns most often, but not always, leads to energy savings and the improvements obtained by individual refactorings could be significantly reduced when combined. While in some cases it was observed that combinations lead to increased energy consumption.

Summary

Looking at the related work we see that our focus area of concurrency has been explored prior to this, however, the concurrency constructs themselves have only been looked at in regard to Java and Haskell. We have from this that time and energy consumption do not seem to correlate, which begs the question of whether this is also true for C#. We can look at the C# counterparts to the language constructs explored in [16].

The papers on generalizability indicate that the impacts of different language constructs might not be consistent across different applications, and depending on what combinations they are used in, their impact may change. However, the research does not look at C#, which is our chosen language. We use these papers as a basis to examine how generalizable language constructs are from benchmark to benchmark and how combining them potentially makes the energy effects differ.

Chapter 2

Language Constructs

In [3] several different language constructs in C# were explored and their effect on energy consumption was found. In this project, we aim to expand the benchmark catalog with benchmarks concerning not previously tested C# language constructs. They are to be analyzed regarding the reasoning behind observed effects and their effects in a larger context with macrobenchmarks. To get a better understanding of the language constructs we test, we explore them in regards to how they are utilized. We present the constructs and what groups they belong to, to give an overview of which constructs are compared.

2.1 Construct Selection

Language constructs are grouped in accordance with their functionality and what constructs they are compared to. When constructing these groups the constructs must have equivalent functionality or outcomes after utilization. This is not always the case, an example of this being the temporary lifetime of a `lambda` function compared to other functions. Also, implicit casting can be accomplished by explicit casting, but not the other way around. However as the same outcomes are possible, and as the language constructs can serve the same functionality, we consider them to be members of the same group.

Group	Constructs
Concurrency	`Task`, Manual, `ThreadPool` and `Parallel.For`
Casting	implicit, explicit and `as`
Parameter Mechanisms	`in`, `ref` and `out`
Lambda	expression, statement, delegate and action

Table 2.1: Language constructs groups and the construct that are examined within a given group.

We see in Table 2.1 the groups of language constructs we explore, and the language constructs we consider members of these groups. We have concurrency as our primary focus, but as this is not self-contained we also look at other advanced language constructs such as parameter mechanisms and lambda expressions which are sometimes used together with concurrency. The members are to be tested against each other to determine the best option in a given group when a developer is primarily concerned with the energy consumption of their software.

Concurrency

The preferred way to write parallel code in C# is by using the Task Parallel Library (TPL), which simplifies the process for developers by automatically scaling the degree of concurrency to the number of available processors, as well as partitioning the work, scheduling it on a thread pool, and other low-level details [23].

It is also a possibility to manually manage a `Thread` and have full control of its lifetime [24]. When using this approach, some concerns must be kept in mind. It is imperative to ensure that threads are started, which requires the use of a delegate to represent the method meant for execution. For synchronization, the threads must also be joined once they terminate.

There is also the option of using a parallel `for` loop, with `Parallel.For` [25]. Using this allows the different iterations of the loop to run in parallel. Alongside this, it also allows for the state of the loop to be monitored and manipulated if so desired. It is used when performing the same independent operation for each element in a collection or a predetermined number of iterations [26].

Casting

In C# there are several ways of casting. We explore the three options presented in Table 2.1. Determining which type of casting consumes the least energy helps to determine which option should be used if it does not hinder or alter the code. This is further motivated by one of the results seen in [3, p. 119-120]. It was found that because of casting and subsequent unboxing one language construct consumed more energy than other constructs in its group [27]. On this basis, we look into the different casting approaches and their effects on energy consumption.

To start we look into implicit casting [28]. This type of casting, as the name specifies, is implicit and is done if two types are compatible and when the value can be converted without loss of information. When it comes to reference types implicit casting can be used to cast from a class to any of its base classes or interfaces, this works because a class derived from another contains all the members of the base class.

Next, we look at explicit casting [29]. This type of casting is done in the form `(T)E`. Expression `E`'s result is explicitly converted to type `T`. This is done in cases where the conversion from one type to another might lead to a loss of information. Explicit casting is also used for reference types [30]. Explicit casting is used when casting from a base class to a derived one, however, this can throw an exception at run time if casting fails. To avoid exceptions there is the option of using the `as` keyword.

Using the `as` keyword [31] allows for explicit conversion to a reference or nullable value type, and is used in the form `E as T`. Using this keyword returns the desired type if it is compatible and otherwise `null`. Another effect of `as` is that it does not throw an exception, but rather returns `null` in cases where casting is not possible.

Parameter Mechanisms

C# allows for parameters to be passed by value or reference. To help facilitate this C# presents three modifiers or keywords, which control how a parameter is passed. The three options are `in`, `ref` and `out`. Each sets up rules for how the argument can be modified and what state it should be in when passed to a method [32].

When using the `in` keyword the parameter argument is passed by reference, and it ensures that the parameter argument is not modified by the called method, it is a read-only reference [33]. One requirement of this

keyword is that the parameter argument must be initialized before being passed to the called method.

The second option presented is the `ref` modifier [34]. As with the other options, the parameter argument is passed by reference and the argument may be modified by the called method, however, it is not a requirement. As with `in`, the parameter argument must be initialized prior to being passed to the called method.

Lastly we have the `out` modifier [35]. Once more the parameter argument is passed by reference, however, unlike the two other options, the arguments have to be modified. Unlike `in` and `ref`, the arguments need not be initialized prior to passing to the called method, however it is a requirement that a value is assigned to the argument before the called method terminates.

Lambda Expression

In C# it is possible to create anonymous functions by using lambda expressions. This language construct serves as our extension to the *Invocation* group from [3, p. 25]. A lambda expression can be of two forms. One is an expression lambda which has an expression as its body. The second is a statement lambda which has a statement block as its body. Any lambda expression can be converted into a delegate type [36, 37]. A delegate defines a type that represents a reference to a method with a particular parameter list and return type. A method whose parameter list and return type matches can be assigned to a variable of that type. The method may then, given the right arguments, be called directly or passed as an argument to another method. To ease the use .Net includes a set of delegate types. These types are `Func`, `Action`, and `Predicate`, which can be used without defining new delegate types[37]. Microsoft states that Lambda expressions are another way of specifying a delegate. We test if that is true or if there is a difference in the energy consumption of delegates and lambda expressions. Determining the energy consumption of lambda expressions compared to different types of invocation will help determine whether to use lambda expressions, delegates or to invoke methods.

2.2 Summary

We examined four different groups of language constructs, which are seen in Table 2.1. Members of the groups are presented showing their purpose for our benchmarks and what they are compared against. Of the four groups, three have been made to explore more advanced C# language constructs, while the Casting group has been chosen to help explain how unboxing resulted in a language construct performing worse than other members of its group [3].

Chapter 3

Design

To extend the information on language constructs and their effect on energy consumption presented in [3], we create benchmarks for the four groups presented in Table 2.1. We utilize the same methodology as presented in [3], however, we introduce changes to parts of the methodology, such as performing sanity checks to determine the reliability of the generated results.

Furthermore, the process for analyzing the reason for differing energy efficiency of language constructs is extended to more reliably determine the reason behind these differences.

We look at macrobenchmarks to acquire knowledge on how different language constructs affect the energy consumption of a larger codebase and how they work in conjunction.

We establish a methodology for determining the generalizability of the results found for the different language constructs across different applications and language construct combinations if such is observed.

3.1 Benchmarks

We have chosen to look at two different categories of benchmarks, these two categories being micro- and macrobenchmarks. The microbenchmarks serve the purpose of extending the knowledge of language constructs presented in [3] by looking at more advanced language constructs in C#.The macrobenchmarks are used to determine if changing applications to use more energy-efficient language constructs, according to [3] and the constructs tested in this project, improve the energy efficiency. Furthermore, they are also used for determining if the outcomes of changing language

constructs are consistent across different benchmarks and language construct combinations.

Microbenchmarks

First, we establish what is meant when talking about microbenchmarks. A microbenchmark is a type of benchmark which tests a single task or component [38]. This also means that microbenchmarks can be of varying sizes. A benchmark concerned with an algorithm and a benchmark concerned with only a single language construct are both considered microbenchmarks. In this project, we look at microbenchmarks which look at a single language construct. Because we use microbenchmarks we also test a well-defined variable [39], which in our case is the energy consumption effects of the given language construct in the benchmark. Utilizing microbenchmarks has both advantages and disadvantages. They allow us to test specific language constructs while keeping other variables as constant as possible so the gathered results can be better used to determine which language construct is better among the ones they are tested against. On the other hand, there is a disadvantage in how they do not give us knowledge on how the different benchmarks affect an application. They also do not allow us to say anything about the generalization of their effect on applications, that is microbenchmarks do not tell anything about how often a language construct is used or how combinations with other language constructs affect it. To obtain this knowledge, as well as answer the questions from Section 1.1, we also look at macrobenchmarks.

Macrobenchmarks

When referring to macrobenchmarks it usually means a benchmark that is used to test an entire application [38], however, the macrobenchmarks we utilize in this project do not look at entire applications, but rather larger benchmarks that utilize more than one language construct to complete a task. Using macrobenchmarks means that, while we can still test for a well-defined variable, which in our case is energy consumption, the effects of changing a single language construct are likely to be obfuscated. This could be considered a disadvantage of using macrobenchmarks, but seeing as we also have microbenchmarks both from our project and [3] which show the effects of individual language constructs, this information is not

lost. Using macrobenchmarks allows us to see the effects of using the information gathered so far on language constructs in C# and how it can affect applications where constructs are used multiple times and influence one another.

3.2 Goals

In this section, we present the goals for the two types of benchmarks we perform, micro- and macrobenchmarks.

Microbenchmarks

In regards to the goals for our microbenchmarks, we have two goals for the knowledge we want to gather. The first goal of our microbenchmarks is to expand upon the knowledge presented in [3], and determine which of the language constructs from each of the groups presented in Table 2.1 is considered the best in regard to energy efficiency.

The second goal of our microbenchmarks is to determine the cause of the different energy efficiency between language constructs within the same group. We utilize the result analysis process presented in [3, p. 111-112] to accomplish this. As all results could not be explained using this process, we extend the steps and construct a process that can more reliably determine the causes for differing energy behavior between similar language constructs.

Macrobenchmarks

The goals for our macrobenchmarks are different from the microbenchmark goals. Using macrobenchmarks we have two goals for the information we seek to obtain. The first goal of utilizing macrobenchmarks is to determine if changing language constructs according to the information gathered in [3] and the information gathered about the language constructs in Table 2.1, lead to increased energy efficiency on an application level. In a continuation to this goal, we look at how combinations of different language constructs affect energy efficiency. It might be the case that some combinations lead to an overall decrease in energy efficiency or no change at all, and if that is the case we look into why. We can have cases where combinations are better than the sum of their parts, and should such cases occur we explore why.

Our second goal for performing macrobenchmarks is to determine the generalizability of the language construct effects across multiple applications. This regards both individual language constructs and combining them with others. Doing this gives us knowledge of whether the effects across applications remain the same. It might be the case that energy efficiency only improves in some applications, and if this is the case we should investigate what causes it.

3.3 Experiment Protocol

In this section, we present how we measure our benchmarks and what performance parameters are examined. Also, we present an auxiliary tool used to help with the analysis of our results. Lastly, we present the hardware setup utilized to conduct our research.

We use the framework presented in [3]. The framework allows us to measure, *Package Energy*, *DRAM Energy*, *Elapsed Time* and *Temperature* of our benchmarks.

We choose to focus on *Package Energy* and *Elapsed Time*. We do this because we are mainly concerned with the energy consumption of our benchmarks, while *Elapsed Time* is important as it allows us to see the effect on runtime, which can be a key factor for time-critical applications. Additionally, we choose these metrics as they are the same as those used in [3], which supports easier comparing of old and new microbenchmark results.

To get a representative sample of measurements, we use the approach recommended in [40] and used in [3, p. 37-39]. We use this methodology to both create and run usable and reliable benchmarks.

Notably, [40] suggests that we should:

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

In regard to the macrobenchmarks, we perform similar measurements to the microbenchmarks. But to be able to compare the energy differences we initially measure an unmodified version of a benchmark to provide the baseline energy consumption. Afterward, the modified variations of the macrobenchmark are run and the results are saved for each permutation to allow for comparison.

However, before the framework is utilized we have elected to perform sanity checks on the results from [3], which allows us to better establish how reliable the results generated by the framework are. The formula used for calculating the rough estimate of consumed energy is seen in Equation 3.1.

$$T \text{ ms} \cdot 1000 \cdot B \text{ Watts} = T \ \mu\text{s} \cdot B \text{ Watts} = C \ \mu\text{J} \tag{3.1}$$

In Section A.1 the sanity checks of the results gathered by the framework in [3] are seen. We find that, under the assumption that the CPU draws 15,83 Watts per core, the difference between what energy consumption is expected given elapsed time and the power draw of the CPU is on average 24,04%, with the largest observed difference being 56,99% and the smallest 3,21%. We deem the framework reliable because results do not differ more than 100% from the rough estimate. This large level of difference from the expected results is allowed as the estimate does not know the actual power draw of the CPU or take into account idle energy consumption from other cores.

Another part of our look into the reliability of the framework focuses on the variance of the benchmark results. In Section A.4 we see the variance experiments and their results. We find that the approximate 1% variance in results [3, p. 166] no longer holds, which is likely caused by updates to the framework. For the benchmark used to present the variance, we instead find a variance of 1,74%. This change leads to testing all of the benchmarks from [3], where we find an average variance of 7,57% and a maximum and minimum variance of 28,61% and 1,38% respectively. While this is different than the initial assumption of results variance, we still deem the framework reliable, however, these numbers must be taken into account when determining the comparative performance of language constructs.

We use a tool called *PowerUp* [41], which allows a user to see the Intermediate Language (IL) code and assembly code of their programs in *release* mode. *PowerUp* is a collection of different utilities and tools, ranging from productivity to decompilation and disassembly. The tool we use is its *Live IDE Watcher*, which allows us to monitor source code files, compile them,

and later decompile and disassemble them to produce IL and assembly code outputs. These outputs are updated live, so changes to the monitored source code are reflected in the IL and assembly.

To ensure consistency for the results of both micro- and macrobenchmarks we use a single computer, with the specification shown in Table 3.1.

Processor	6 core 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.

3.4 Guidelines

In Section 3.3 a number of guidelines were established, which help ensure consistency for the benchmarks. Following these guidelines makes our benchmarks consistent with the ones presented in [3], thereby making them comparable.

Utilizing the framework [3] ensures we follow most of the guidelines presented as well as taking care of determining the number of iterations of each benchmark to get significant results, we need only take one special consideration into account for our microbenchmarks. One thing we must consider when writing our benchmarks is the tip of saving results inside the benchmark to a dummy variable from [40], which helps ensure that the compiler does not optimize it away.

One thing to note here is that the guidelines are designed for microbenchmarks. Therefore depending on the type of benchmark some of the guidelines presented are of lesser concern when looking at macrobenchmarks. The iterations needed within a benchmark are handled by the framework and as such, it also works for an application with an execution time below 0,25 seconds. As the framework increases the number of loop iterations if the benchmark takes less than 0,25 seconds[3].

Writing Benchmarks

We now set up the process for writing micro- and macrobenchmarks. The process for microbenchmarks follows what was presented in [3, p. 44], however, we modify some parts. Since the macrobenchmarks are comprised of applications we present the process used to change them, as to test the effect of altering different language constructs.

Microbenchmarks

Writing microbenchmarks follows the process shown in Figure 3.1. First, the initial benchmark is created and then reviewed by peers, being the project members, and depending on the outcome, the benchmark is either modified and then reviewed again or we move to the next phase. When the benchmark passes the peer review we then check if it gets compiled away. If so, we modify and review again, otherwise, the benchmark is accepted.

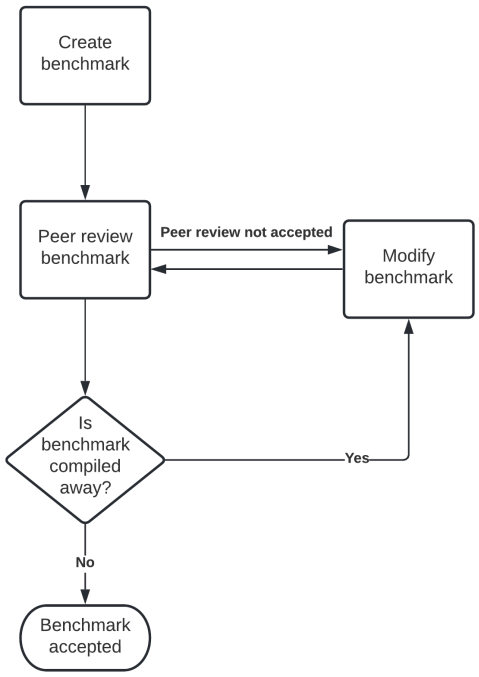

Figure 3.1: Microbenchmark writing process.

To check if a benchmark has been compiled away we check the assembly code. This simplifies the process presented in [3], which depended on

comparing benchmark measurements against an empty `for` loop.

Macrobenchmarks

The process of writing macrobenchmarks involves changing a benchmark to use more energy-efficient language constructs. The macrobenchmarks are constructed using the information gathered in the microbenchmarks constructed in this project as well as the result for the different microbenchmarks in [3]. For the initial benchmark which tests the energy consumption and elapsed time of the original macrobenchmark, little to no changes are required. For the variations that look at language constructs, the benchmark is modified with the use of one language construct. If the behavior is altered, changes are reverted as changing the used language constructs should not alter behavior. This means that in some cases the best performing language construct from a group may not be applicable. Once a macrobenchmark has been altered and its behavior remains the same it is accepted. For each of the macrobenchmarks, we use three different language constructs. This means we create seven permutations, as first, we have benchmarks that use one language construct, then two language constructs, and lastly three ($\frac{3!}{1!(3-1)!} + \frac{3!}{2!(3-2)!} + \frac{3!}{3!(3-3)!} = 7$). Taking the original version into account we end up with eight benchmarks for each macrobenchmark.

3.5 Threats to Validity

We look at four different categories in regard to the threats to the validity of our results. These are *Construct Validity*, *Internal Validity*, *External Validity* and *Reliability*, and are inspired by [42]. Looking at these categories allows us to better reflect on the results gathered from our experiments.

Construct Validity

Construct Validity is whether we measure what we set out to test or not. In our case, we want to measure the energy consumption and elapsed time of our micro- and macrobenchmarks. In order to ensure this, we make use of the framework and guidelines from [3]. The major concern here is ensuring that our benchmarks are not optimized away, this is handled by using the tool *PowerUp*, which allows us to see the release version of the assembly code for our benchmarks as they are being written. Because of this, we

can determine if part of a benchmark is compiled away and change it until *PowerUp* no longer shows that the benchmark is compiled away.

Internal Validity

Internal Validity concerns whether or not other factors/variables influence the results of our benchmarks. There are several concerns that could impact the results, ranging from CPU temperature, OS context-switching, garbage collection, CPU frequency scaling to system daemons. Temperature is handled by monitoring the temperature of the CPU at the start and end of a benchmark and allowing us to determine if results were gathered at the same temperature range. Context switching is handled following the guidelines from [3] and [43], which dictate that enough measurements should be taken to ensure significant results, and cores should be shielded to ensure that uninvited threads do not run on the cores used for benchmarking.

External Validity

External Validity concerns itself with whether or not the results can be generalized to other programming implementations. The concern here is if the results of our microbenchmarks can be generalized to applications. As one of the goals of this project is to determine the generalizability of macrobenchmarks, which can be small applications, and as such this particular threat to validity is not a major concern. As we test multiple applications, the risk of reduced external validity should be heavily mitigated.

Reliability

With *reliability*, we concern ourselves with the reproducibility of the results form our benchmarks. This is mitigated both through the guidelines we use and the use of the measurement framework [3].

Some major concerns in regard to reliability are; random context switching, clock speed, and randomness of hardware states. These concerns are mitigated by following the methodology used in [3], as the same concerns were faced there, and additionally using the tips from [43]. The tips are among others:

- shield CPUs from uninvited threads, so the CPU is not used by something else during benchmarks,

- making the clock rate effectively static and turning off boosting the CPU,
- turning off hyperthreading ease CPU resource management, and
- ensuring interrupt requests are not sent to cores used for benchmarking.

3.6 Summary

This section presents the goals of our benchmarks as well as the design that helps us achieve these. We utilize microbenchmarks to extend the research in [3] both concerning the energy effects but also to improve upon the process for analyzing the reason for a language constructs behavior. Macrobenchmarks are used to determine if the results from the microbenchmarks are generalizable to applications and to see if the effects of language constructs are consistent when multiple language construct groups are altered in the same macrobenchmark. We use the framework [3] to measure our benchmarks. We focus on the metrics for *Package Energy* and *Elapsed Time* when looking at the information gathered. This is done for both the micro- and macrobenchmarks. We have deemed the framework reliable based on our sanity checks, seen in Section A.1. We utilize the guidelines from [3], which are partly based on [40]. Using the framework takes care of most of these guidelines and gives the added benefit of automatically determining the number of iterations needed to get significant results for our benchmarks. The different threats to validity are described and for each of the categories, we present the potential threats and ways in which they are mitigated.

Chapter 4

Microbenchmarks

All microbenchmarks use the setup presented in Section 3.3, Table 3.1, furthermore all results are normalized to 1.000.000 iterations, as per the framework [3, p.60].

4.1 Results

In this section, we present our microbenchmarks and their results. All the benchmarks can be found in our git repository [1]. We have constructed the benchmarks ourselves because the utilized framework requires the benchmarks to follow a specific structure, where they must take no input parameters and must return something. This allows for comparison with other benchmarks, where we can ensure that benchmarks return the same things and are thereby comparable. Furthermore, the benchmarks are constructed in such a manner that the work which is central to the measurement must be part of a loop inside the benchmark method. Based on these requirements and looking at related literature we have not found benchmarks that are suitable for our specific testing purposes and as such we have elected to construct them ourselves.

Presentation of the results is done by first describing what is tested and what variations exist, this is accompanied by an explanation of any subgroups that may exist in the larger groupings. These groupings are created based on the information presented in Chapter 2.

Next, we show a benchmark to establish an overview of what benchmarks for language constructs look like in the group. The example benchmarks are then followed by an explanation of both themselves and their

differences from other benchmarks in the group.

We then show plots of the results, in these plots, the dashed line is the average and the solid line is the median. The boxes in the plots show the 95% interval of the results, while the outermost lines show the biggest outliers. If only one of average or median is shown, it is because they are so similar if not identical, that they are placed at the same point in the plot.

Following this, we present an overview of how the results look across 10 runs. We do this based on our findings from Section 3.3 and Section A.4 about the variance of benchmarks between different runs. Running the benchmarks 10 times has the purpose of gathering a sample of results, which we use to determine whether or not two language constructs are from the same population. We do this by conducting Two-sample Kolmogorov–Smirnov tests [44]. This is accomplished through the use of the accord implementation `TwoSampleKolmogorovSmirnovTest` [45]. The results of the test are seen in Section A.8. The 10 runs also allow us to see how much each benchmark varies in regard to energy consumption, which allows us to judge the comparative energy efficiency of language constructs. 10 runs of the benchmarks are chosen based on the execution time of the benchmarks. We have found that running the concurrency benchmarks 10 times takes between 72 and 96 hours and because of this, we elect to not run the benchmarks more than this.

We then present a table containing the numeric values for all of the recorded metrics for a group of benchmarks, these being *Package Energy*, *DRAM Energy*, *Elapsed Time*. The numbers in these tables are color-coded, with the lowest in a column being marked with blue and the highest with red. We use the numbers presented in that table in conjunction with Equation 4.1 to determine the percentage difference in energy consumption. This difference is then used to determine if a language construct is better than other options.

$$\frac{|V_1 - V_2|}{\left[\frac{(V_1 + V_2)}{2}\right]} \cdot 100 = Percentage\ Difference \tag{4.1}$$

The difference needed is determined on a case-by-case basis, based on the variance observed in the results, this decision is made based on the findings presented in Section 3.3. Lastly, we perform sanity checks for our results using Equation 3.1 from Section 3.3. The sanity checks are done under the assumption that the power draw of a single core of the Intel Xeon W-1250P has a power draw of 15,83 watts, as presented in Section 3.3 and

Section A.1. Because of this assumption, the estimated values are different from the recorded results. We evaluate if the difference is plausible, by checking if the difference between the estimate and recorded value is below 100%.

The sections about a group of language constructs are then ended by establishing an outline of what the findings are.

4.1.1 Concurrency

This section presents the results of our concurrency microbenchmarks. We have four different types of concurrency benchmarks, which are presented in Chapter 2. Each of these has several different variations as to benchmark differing number of threads used and changing workloads.

- `ParallelFor`
- `UsingTasks`
- `ManualThread`
- `ThreadPool`

Beyond these, we also have sequential benchmarks performing the same work. The `sequential` benchmarks allow us to determine the comparative energy consumption of concurrent and sequential execution. Furthermore, the concurrency benchmarks have four cores available when run, with hyperthreading turned off as per Section 3.5. This allows us to determine how different degrees of concurrency affect energy efficiency. To compare the constructs for concurrency we ensure that the same work is accomplished by all of the benchmarks by using the method in Listing 1.

```
1  public static class SharedResources {
2      ...
3      public static ulong DoSomeWork(ulong amountOfWork) {
4          ulong a = 2, b = 2;
5          for (ulong i = 0; i < amountOfWork; i++) {
6                  if (a == 0) {
7                          a = 1;
8                  }
9                  a *= b;
10                 b++;
11         }
12         return a;
```

```
13      }
14      ...
15  }
```

Listing 1: The method used by the different concurrency construct benchmarks, to ensure that the same work is accomplished.

In Listing 1 the method takes the argument `amountOfWork` specifying how many iterations are executed, which gives two benefits. First, it helps mitigate the risk of code being compiled away, as the code is dependent on this argument. The other benefit is that it eases the testing of concurrency constructs with differing workloads. We run all the benchmarks presented in this section with the values 1, 10, 100, 1.000, 10.000, 100.000, 1.000.000, and 10.000.000 for `amountOfWork`. We increase the value by multiplying the previous value by 10. By creating this span of values we intend to find how the energy efficiency of the benchmark behaves as the amount of work increases. This gives us insight into both the minimum amount of work needed for a concurrent approach to be better than sequential, but also if there is any point along these values where a particular language construct becomes better or worse than the alternatives.

Given this information, we explain how the `Parallel.For` benchmarks are structured.

```
1   public class Concurrency1K {
2       ...
3       [Benchmark("Concurrency1000", "Tests simple Parallel.For")]
4       public long ParallelFor2t() {
5               long res = 0;
6               Parallel.For<long>(0L, (long)LoopIterations, new
                ↪  ParallelOptions { MaxDegreeOfParallelism = 2 },
                ↪  () => 0, (_, _, subtotal) => {
7                               subtotal += (long)DoSomeWork(1000);
8                               return subtotal;
9                       }, x => Interlocked.Add(ref res, x)
10              );
11              return res;
12      }
```

```
13      ...
14  }
```

Listing 2: The benchmark which tests `Parallel.For` when using a maximum of two threads.

In Listing 2 the benchmark structure for the `Parallel.For` benchmarks is seen. The variable `subtotal` is a thread local variable. This means that iterations that run on the same thread use this variable that is passed between iterations of the loop. When the loop terminates, the subtotal from each thread is then added to `res` with `x => Interlocked.Add(ref res, x)`, where `x` contains the value from `subtotal`. The `Interlocked.Add` function ensures an atomic add operation. This version of the benchmark has been allowed a maximum of two concurrent operations, or threads, by setting `MaxDegreeOfParallelism = 2`. This value changes between the different variations of which there are three others, being 3, 4, and 6. These values were chosen as our initial look into the construction of concurrency benchmarks. The initial design idea what to create a version for each amount of possible threads, which were available, and one benchmark which is allowed to use more threads than available. There is another variation, which excludes `MaxDegreeOfParallelism`, which is the default behavior of `Parallel.For`. The default behavior uses as many threads as available [46]. All these variations are used to perform the task defined in `DoSomeWork()`, Listing 1.

```
1   public class Concurrency1K {
2       ...
3       [Benchmark("Concurrency1000", "Tests creating threads
    ↪   manually")]
4       public ulong ManualThread2() {
5           const ulong numThreads = 2;
6           Thread[] tArr = new Thread[numThreads];
7           ulong[] resArr = new ulong[numThreads];
8               ulong iter = LoopIterations / numThreads;
9               ulong firstIter = LoopIterations - (iter *
                ↪  numThreads) + iter;
10              for (ulong j = 0; j < numThreads; j++) {
11                      ulong j1 = j;
```

```
12                      if (j == 0) {
13                              tArr[j] = new Thread(() => {
14                                      ulong temp = 0;
15                                      for (ulong i = 0; i <
                                        ↪  firstIter; i++) {
16                                              temp +=
                                                ↪  DoSomeWork(1000);
17                                      }
18                                      resArr[j1] = temp;
19                              });
20                              tArr[j].Start();
21                              continue;
22                      }
23                      tArr[j] = new Thread(() => {
24                              ulong temp = 0;
25                              for (ulong i = 0; i < iter; i++) {
26                                      temp += DoSomeWork(1000);
27                              }
28                              resArr[j1] = temp;
29                      });
30                      tArr[j].Start();
31              }
32      ...
33  }
```

Listing 3: The benchmark which tests using threads manually, in this case with 2 threads.

In Listing 3 we can see how concurrency is handled when manually controlling the threads.

This part of the benchmark shows how we set up the work for each thread, which is then added to an array of threads, and the thread is started. `firstIter` is created in order to make sure all `LoopIterations` are executed if `LoopIterations` is not divisible by `numThreads`. This benchmark is constructed in a generic manner making it possible to use a different amount of threads by changing the `numThreads` variable. This means that the benchmark is more complex than needed to use two manual threads, but it allows for easier construction of other benchmarks and a better comparison of how the number of threads influences energy consumption. There are 8 variations of this benchmark, the one shown in the listing for two threads and

7 others for 4, 6, 12, 24, 48, 96, and 192 threads. Having these variations allows us to see how well the construct scales with the number of threads, and when it surpasses the number of physical cores.

```
1  public class Concurrency1K {
2      ...
3              foreach (Thread thread in tArr) {
4                      thread.Join();
5              }
6              ulong sum = 0;
7              Array.ForEach(resArr, res => sum += res);
8              return sum;
9      }
10     ...
11 }
```

Listing 4: The benchmark which tests using threads manually. Continued from Listing 3.

Listing 4 shows the remainder of the benchmark. The benchmark has a `foreach` loop joining the threads. The results of the threads are then summed to produce a result, which is returned to compare benchmark results.

```
1  public class Concurrency1K {
2      ...
3      [Benchmark("Concurrency1000", "Tests using tasks")]
4          public ulong UsingTasks() {
5                  ulong res = 0;
6                  Task<ulong>[] reArr = new
                   ↪  Task<ulong>[LoopIterations];
7                  for (ulong i = 0; i < LoopIterations; i++) {
8                          reArr[i] = (Task.Run(() =>
                           ↪  DoSomeWork(1000))
9                                  .ContinueWith(task =>
                                   ↪  Interlocked.Add(ref res,
                                   ↪  task.Result)));
10                 }
11                 Task.WaitAll(reArr);
```

```
12                  return res;
13          }
14      ...
15  }
```

Listing 5: The method used by the different concurrency `Task` benchmarks.

Next in the list of concurrency benchmarks we have `UsingTasks` from Listing 5. In this benchmark `Task` is used, by first creating an array for all the tasks. Then in the `for` loop tasks are added to the array and set to run with their results being summed using `Interlocked.Add` after completion, which ensures atomic add operations on the `res` variable. Once all tasks are complete the result is then returned.

```
1  public class ThreadPoolBenchmarks {
2    ...
3      [Benchmark("Concurrency1000", "Tests using thread pool")]
4      public static ulong ThreadPool1000() {
5              _resArr = new ulong?[LoopIterations];
6              for (ulong i = 0; i < LoopIterations; i++) {
7                      ThreadPool.QueueUserWorkItem(o => {
8                              ulong index = (ulong)o!;
9                              _resArr[index] = DoSomeWork(1000);
10                     }, i);
11             }
12   ...
13 }
```

Listing 6: The method used by the different `ThreadPool` benchmarks.

As the last example we have in Listing 6 the use of `ThreadPool`. An array is created to contain the results. Then in our `for` loop, work is being queued on the thread pool. The index in the loop is passed to the anonymous function, such that the result can be added to that index of the results array.

```
1  public class ThreadPoolBenchmarks {
2      ...
3      public static ulong ThreadPool1000() {
```

```
4       ...
5               ulong res = 0;
6               for (ulong i = 0; i < LoopIterations; i++) {
7                       while (!_resArr[i].HasValue) {
8                               Thread.Sleep(0);
9                       }
10                      res += _resArr[i]!.Value;
11              }
12              return res;
13      ...
14  }
```

Listing 7: The method used by the different `ThreadPool` benchmarks. Continued from Listing 6.

Listing 7 shows the last part of our `ThreadPool` benchmark. We loop through the result array and wait until a value is present in our current cell, and the results are then summed.

Benchmark results

We start by looking at the results with the lowest amount of work, which is 1 iteration of `DoSomeWork`, and then the one with the most work, which is 10.000.000 iterations of `DoSomeWork`. We have chosen this split as it gives an example of how the cost of different concurrency constructs is dependent on the amount of work they are being used for. It shows how some concurrency constructs need a certain amount of work before they are preferable to a sequential approach. Following the presentation of these amounts of work, we then present an overview of the results of the different concurrency constructs across all the differing amounts of work.

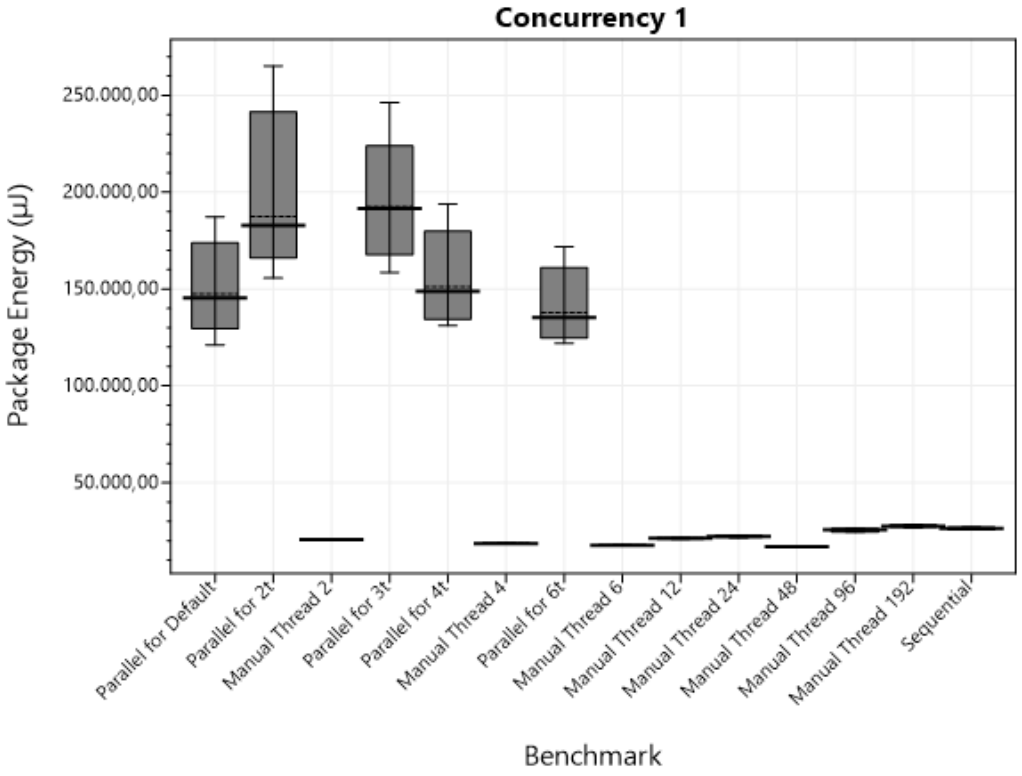

Figure 4.1: The energy consumption of the different approaches for concurrency, with little work. The y-axis starts at 0 and ends at 280.000μJ.

In Figure 4.1 we see a comparison between the different concurrency constructs when `amountOfWork` is set to 1, which means very little work is being performed per iteration of the loop.

To note is that the benchmark using `ThreadPool` and the `Tasks` library have been removed from the plot, as they consume much more energy as can be seen in Table 4.1. For 1 iteration of `DoSomeWork`, across 10 runs, `ManualThread4` is best four times, `ManualThread6` is best once, `ManualThread12` is best twice, `ManualThread24` is also best twice, and `ManualThread48` is best once. Variance across these 10 runs is seen in Table A.72 in Section A.5. We see from this that using manual threads is consistently the better approach. Looking at the results of the Kolmogorov-Smirnov tests in Table A.106, Section A.8.1, we see that in most cases the benchmark results are from different probability distributions. However we can see that `ManualThread4` and `ManualThread6` are part of the same distribution, and the same goes for `ManualThread12` and `ManualThread4`, which supports the results observed across 10 runs.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Parallel for Default	4,08	147.401,74	2.303,33
Parallel for 2t	5,30	187.491,30	2.969,24
Manual Thread 2	1,10	20.654,71	613,25
Parallel for 3t	5,23	192.405,12	2.944,12
Parallel for 4t	4,12	151.222,15	2.322,06
Manual Thread 4	0,55	18.486,50	306,37
Parallel for 6t	3,74	137.710,44	2.113,50
Manual Thread 6	0,46	17.626,33	253,92
Manual Thread 12	0,58	21.181,99	320,75
Manual Thread 24	0,62	22.105,55	343,23
Manual Thread 48	0,44	17.022,94	245,99
Manual Thread 96	0,75	25.600,85	418,06
Manual Thread 192	1,02	27.524,67	569,11
Using Tasks	445,47	8.784.535,23	497.133,33
Thread Pool 1	145,5	3.559.783,97	137.453,66
Sequential	2,14	26.316,46	1.188,61

Table 4.1: Table showing the elapsed time and energy measurement for each Concurrency 1 benchmark.

Looking at Table 4.1, and accompanying p-values in Section A.2.1, we see that with this amount of work, only the benchmarks using manual threading, except for the one using 192 threads, are more energy-efficient than doing the work sequentially. This is consistent across 10 runs and surprising as we assumed that the overhead of making these threads would be more costly for a low amount of iterations than the gains in terms of time and energy consumption. `Parallel.For` approaches use about 10 times more energy than the more efficient manual threading and sequential approach.

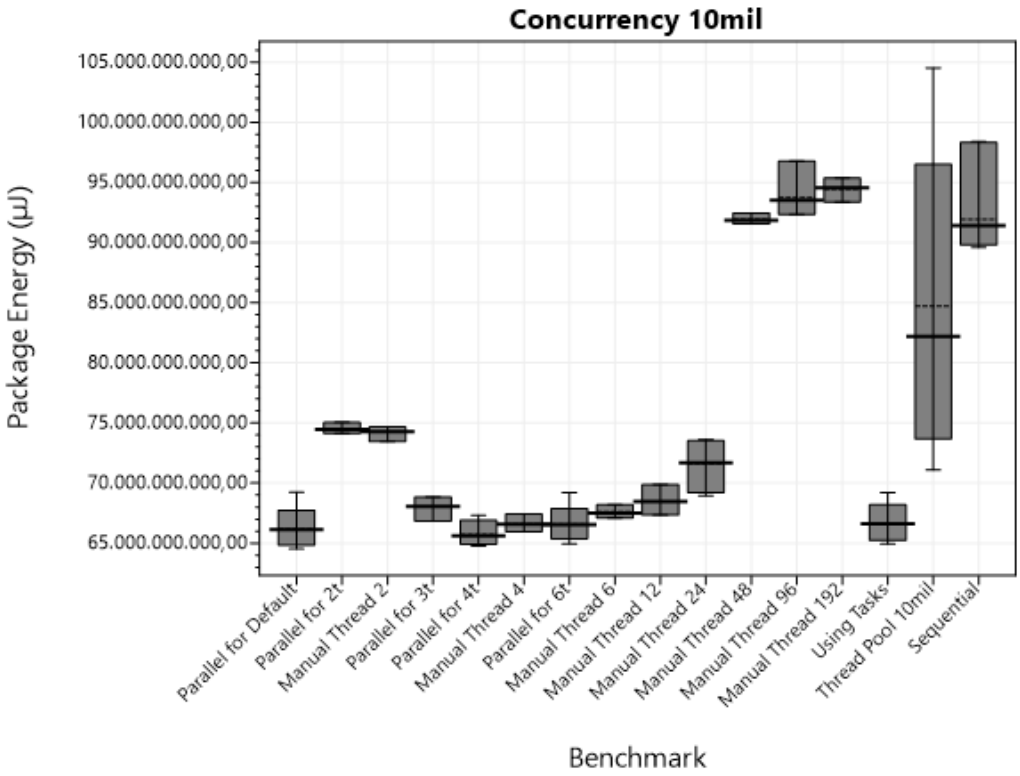

Figure 4.2: The energy consumption of the different approaches for concurrency, with much work. The y-axis starts at 62 billion μJ and ends at 107 billion μJ.

In Figure 4.2 we see a comparison between the different concurrency constructs when `amountOfWork` is set to 10.000.000, which means a much larger amount of work is being performed in each iteration of the loop.

From these results we see that `Parallel.For` outperforms or has similar performance as using manual threading or using the `Tasks` library. Using `ThreadPool` still consumes much more energy than `Parallel.For` or the `Tasks` library, and the `ThreadPool` varies greatly as seen in the plot. For 10 million iterations of `DoSomeWork` across 10 runs, `ParallelForDefault` is best four times, and `UsingTasks` is best six times. This variation between which language construct is better, alongside the recorded values leads us to conclude that `ParallelForDefault` and `UsingTasks` are comparable choices when it comes to energy consumption. We also find that when using 48 threads or above the performance significantly decreases. This is expected as the operating system is likely to use much more time context switching between threads. It is however interesting that the performance of the benchmarks using 48, 96, or 192 threads do not differ more than

they do, and why there is such a large jump between the benchmarks using 24 and 48 threads, considering only four cores are available. The variance of these 10 runs is seen in Section A.5, Table A.73. When looking at the Kolmogorov-Smirnov tests in Section A.8.1, Table A.113 one thing to notice is how `ParallelForDefault` and `UsingTasks` are seemingly part of the same probability distributions, which fits with their rankings seen across our 10 runs. The similar performance of `ManualThread48` and `ManualThread96` is also supported by the Kolmogorov-Smirnov tests as they are likely from the same probability distribution, this is also the case for `ManualThread96` and `ManualThread192`.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Parallel for Default	2.358.728,36	66.172.046.205,36	1.312.102.619,05
Parallel for 2t	4.502.227,34	74.464.989.062,50	2.503.675.000,00
Manual Thread 2	4.511.145,31	74.226.570.312,50	2.508.634.375,00
Parallel for 3t	3.080.954,33	68.003.552.884,62	1.712.023.437,50
Parallel for 4t	2.305.857,84	65.715.240.734,01	1.282.400.314,92
Manual Thread 4	2.286.988,36	66.585.275.000,00	1.271.530.468,75
Parallel for 6t	2.410.649,09	66.572.445.865,39	1.340.457.764,42
Manual Thread 6	2.384.451,52	67.612.737.500,00	1.326.200.000,00
Manual Thread 12	2.397.649,92	68.471.900.971,28	1.334.251.266,89
Manual Thread 24	2.516.085,26	71.625.297.607,42	1.400.064.860,03
Manual Thread 48	9.012.848,75	91.907.265.625,00	5.022.415.625,00
Manual Thread 96	9.023.023,44	93.733.633.522,73	5.043.796.875,00
Manual Thread 192	9.014.904,06	94.444.796.875,00	5.036.915.625,00
Using Tasks	2.314.745,14	66.595.998.104,98	1.287.530.195,22
Thread Pool 10mil	2.772.605,81	84.702.283.208,23	1.542.348.407,58
Sequential	9.001.406,91	91.923.502.878,29	5.006.023.437,50

Table 4.2: Table showing the elapsed time and energy measurement for each Concurrency 10mil.

The numeric results are seen in Table 4.2, and the p-values in Section A.2.1. It becomes apparent that using `Parallel.For` and the `Task` library is inefficient when little computational work is to be done and is more efficient when the computational work is large.

It is interesting to look at when the concurrent construct becomes more energy efficient than the sequential alternative, and all the numbers can be seen in Section A.7. Keep in mind that the following numbers are rough estimates. The `Parallel.For` with 2 threads becomes more energy efficient than sequential at around 10.000 `AmountOfWork`. With 3 threads, this number

is roughly at 5.000 `AmountOfWork`. With 4 threads, this number is roughly at 500 `AmountOfWork`. When using the default for `Parallel.For` this number is around 100 `AmountOfWork`. This makes sense, as the more threads are used, to a limit, the better utilization of resources.

If using the `Task` library the `AmountOfWork` needed is roughly 8.000. This is surprising, as using `Task` is one of the better options when looking at 10 million iterations of work, and at that point, it is also comparable to `Parallel.For`.

If using `ThreadPool` the amount of work needed is roughly 1.000 iterations of `AmountOfWork`, however, `ThreadPool` is generally not as power efficient as the other approaches. We hypothesize this has to do with our specific implementation, as a large array is used for storing the intermediate results from each loop iteration.

For our manual threading approach, the benchmarks using 2, 4, 12, and 24 threads are always more energy efficient than the sequential approach. This is surprising, as our assumption for creating this benchmark was that the initial creation of threads will incur overhead.

Using 48 threads starts out being more energy efficient but at approximately 9.000.000 `AmountOfWork` becomes less energy efficient. Using 96 threads is roughly on par with sequential throughout, and using 192 threads fluctuates between being equal and a bit worse than sequential for all `AmountOfWork`. This is more in line with our expectations, because this is far beyond the number of available cores, and as such we are paying for a large number of threads that are waiting to be scheduled for most of their lifetime.

Looking at Table A.72 and Table A.73 in Section A.5 we see the results in general vary a lot. For 1 iteration the variance is as high as 77,67% for `ManualThread192`, however, when using 10 million iterations the maximum variance is 15,57% in the `ThreadPool10mil` benchmark. Surprisingly, the results of the sequential benchmark vary by 26,27% when running 1 iteration, and 8,23% with 10 million iterations of `DoSomeWork`.

We now perform sanity checks to determine the plausibility of our results. We make use of Equation 3.1 with the assumption that a single core uses 15,83 Watts. First, we check the benchmark `Sequential` with 10.000.000 `AmountOfWork`

$$9.001.406,91 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 142.492.271.352\mu\text{J} \tag{4.2}$$

Equation 4.2 shows the estimated energy consumption of `Sequential` with 10.000.000 `AmountOfWork`, and we find that there is a difference of

43,15% between this value and the value from the benchmark. This is within what we consider plausible according to our procedure in Section 4.1.

Next, we check the plausibility of a benchmark using multiple cores, and `Parallel For Default` is chosen, also with 10.000.000 `AmountOfWork`. As multiple cores are utilized, the estimate is multiplied by the number of available cores.

$$(2.358.728,36 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts}) \cdot 4 = 149.354.680.045 \ \mu\text{J} \tag{4.3}$$

In Equation 4.3 we see the rough estimate and find a difference of 77,19% from the recorded result. We consider the result plausible according to our procedure in Section 4.1. It should be noted that we do not expect this to be an accurate estimate, as all four cores would not draw the same amount of power. The extra cores have less work as they do not run the main process and as such are expected to use less energy overall.

Findings

The results in this section help determine which concurrency construct to use. As expected, using concurrency, in general, entails some overhead. Depending on the size of the work to be executed, and the concurrency construct used this initial overhead can be turned into a significant improvement both in terms of energy and time.

Concurrency constructs can be used in several ways, for instance, our benchmarks using manual threading mimic the behavior of `Parallel.For` where the loop is split into independent segments, but one could have made it more closely mimic the behavior of a `ThreadPool`.

In summary, the results are as follows:

- For small workloads, segmenting the work manually using threads is the most energy-efficient approach, however, as the work becomes larger, using `Parallel.For` or the `Task` library becomes just as efficient as manually using threads.
- The initial overhead of using `Parallel.For` is significantly larger than manually separating the work on multiple threads when each loop iteration executes very little work,
- Our `ThreadPool` benchmark uses significantly more energy than other concurrent constructs throughout varying `AmountOfWork`,

- Energy efficiency when using manual threads significantly decreases between using 24 and 48 threads in large workloads.

4.1.2 Casting

Here we present the results of different methods of casting. The casting constructs are divided into two subgroups, one for casting numeric types and one for casting from a base class to derived classes, which were presented in Chapter 2. There are other types of conversions [47] that casting can be used for, but they are not included in this project, as this group of language constructs is tested in response to the effect of unboxing from [3]. We divide them as not all constructs can be used for the same purpose.

Numeric Casting

For numeric casting we look at two language constructs. As the casting in this subgroup is numeric conversion, specifically a widening from `ulong` to `double` implicit conversion is possible.

- `implicit`
- `explicit`

This is done because explicit conversion covers all implicit conversions, but implicit conversions cannot perform a conversion where there is a loss of data. This behavior is replicated by adding an explicit conversion that does the same as its implicit counterpart.

```
1  public class CastingBenchmarks {
2      ...
3      [Benchmark("Casting", "Tests explicit casting")]
4      public static double Explicit() {
5              double result = 0;
6              for (ulong i = 0; i < LoopIterations; i++) {
7                      result = result + (double)i;
8              }
9              return result;
10     }
11     ...
12 }
```

Listing 8: The benchmark for testing the use of explicit casting for type conversions.

Listing 8 shows the structure for the numeric conversion in our benchmarks. This benchmark casts an `ulong` to a `double`, which is seen inside the `for` loop. We here make use of an explicit cast, even though implicit numeric conversion is possible, this is done to test if there is a difference between explicit and implicit casting. The other variation of this benchmark is identical except for the change of the explicit cast to an implicit one.

Benchmark results

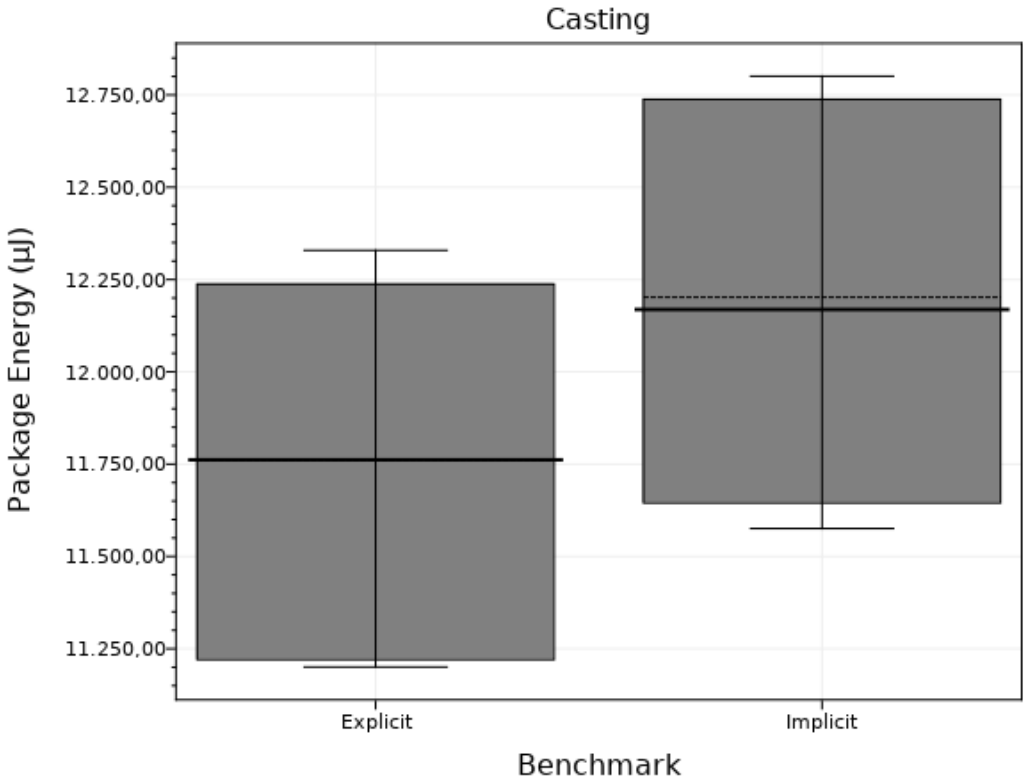

Figure 4.3: The energy consumption of numeric casting. The y-axis starts at 11.150 μJ and ends at 12.850 μJ.

In Figure 4.3 we see the result for the benchmarks in this subgroup.

Figure 4.3 shows that using explicit casts results in lower energy consumption than implicit casts. However, this is not necessarily true, as, over

the course of 10 runs, we have observed that explicit is better in six of the runs, while implicit is better in four of these runs, which we deem to vary too much between runs to draw a conclusion. For a look at the variance exhibited across these runs see Section A.5, Table A.74. We also have from our Kolmogorov-Smirnov tests, that these two language constructs do not stem from different probability distributions, this is seen in Section A.8.2, Table A.114. We explore why this behavior is observed in Section 4.2.2.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Explicit	0,979222	11.757,88	549,49
Implicit	0,979219	12.198,79	549,46

Table 4.3: Table showing the elapsed time and energy measurement for each Casting.

Table 4.3 shows the numeric results for one run of the benchmarks. The p-values for the numeric casting results are seen in Section A.2.2. One should note that based on the changing rankings we have observed across runs, we do not consider these results capable of conclusively determining the better language construct. Looking at Table A.74 in Section A.5, we can see that the two benchmarks have a substantial degree of variance with 17,46% for `Implicit` and 7,01% for `Explicit`. The difference between the consumed energy is 3,68% in this run of the benchmarks. This difference combined with the degree of variance observed and the results of the Kolmogorov-Smirnov tests serves as the reason for the conclusion that we cannot determine which language construct is better. This is not surprising as our assumption was that these language constructs do exactly the same.

We now perform our sanity check to see if the energy consumption recorded in these results is plausible. Equation 3.1 is used under the assumption that a single core uses 15,83 Watts, as presented in Section 4.1 and Section 3.3. We only show one sanity check here as the values are nearly identical for the elapsed time of explicit and implicit casting.

$$0,979222 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 15.501,08\mu\text{J} \tag{4.4}$$

Equation 4.4 shows our estimated energy consumption for explicit casting. This value and the one recorded for explicit casting in our benchmark results have a percentage difference of 27,46 %, which we consider a plausible difference between the actual and estimated energy consumption.

Derived Casting

In this subgroup we have two language constructs:

- `explicit`
- `as`

As it is a requirement that the `as` keyword is used with a reference/nullable type, the benchmarks need to be constructed to reflect this. Therefore, this subgroup looks at explicit reference conversion instead of explicit numeric conversion.

```
public class CastingBenchmarks {
    ...
    [Benchmark("ReferenceCasting", "Tests casting using the as
    ↪ keyword")]
    public static string As() {
            string result = "";
            object o = "Casting";
            for (ulong i = 0; i < LoopIterations; i++) {
                    result = (o as string) + i;
            }
            return result;
    }
    ...
}
```

Listing 9: The benchmark used for testing the `as` keyword used for reference conversion.

The two benchmarks in this subgroup follow the structure seen in Listing 9. In this benchmark we use the `as` keyword to convert an object of class `object` to the derived class `string`. The benchmark for `explicit` is near identical except the conversion from `object` to `string` is done using `result = (string)o + i`.

Benchmark results

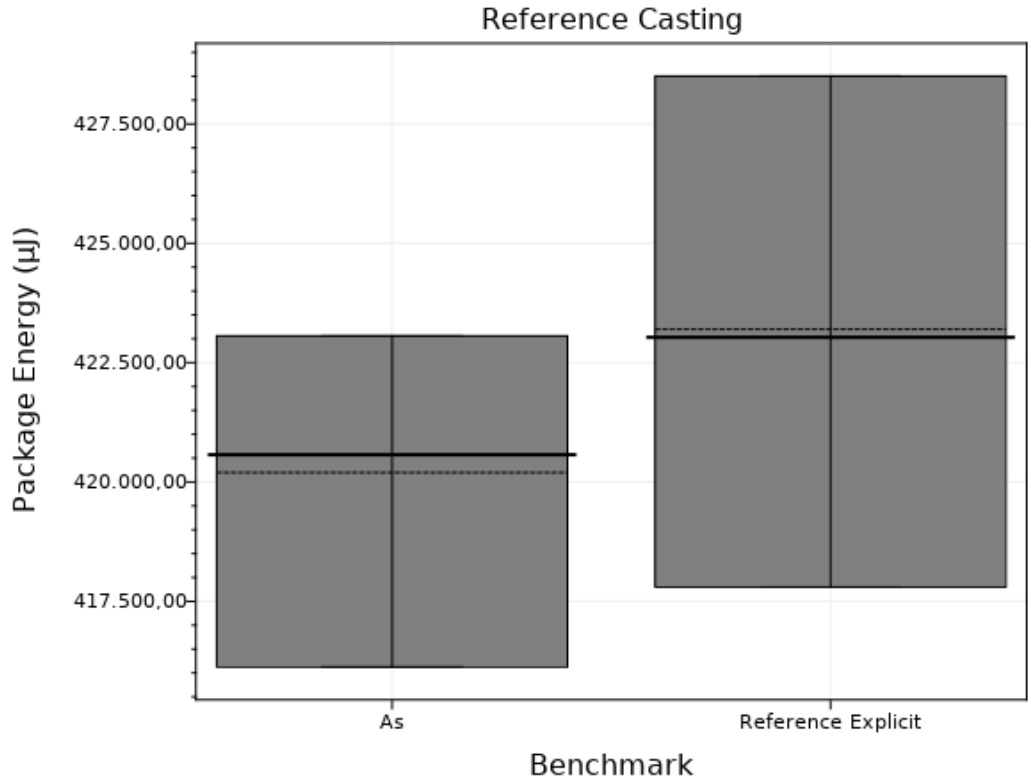

Figure 4.4: The energy consumption of derived casting. The y-axis starts at 415.500 μJ and ends at 429.000 μJ.

Figure 4.4 gives an overview of our results. We see how the `As` benchmark consumes less energy than its equivalent counterpart in `Reference Explicit`. Given 10 runs `as` is the better in 8 of them, which implies that using `as` is preferable to regular explicit casting. However from the results of our Kolmogorov-Smirnov tests we have that they are not from different probability distributions, which suggest that they are similar in behavior and effects, these results are seen in Table A.115, Section A.8.2. We look into this during the analysis in Section 4.2.2.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
As	31,67	420.264,77	21.679,38
Reference Explicit	31,72	423.097,18	21.748,48

Table 4.4: Table showing the elapsed time and energy measurement for each Reference Casting.

Looking at the numerical values in Table 4.4, we see the difference between the two benchmarks. The results are close to one another and as such we check the percentage difference to determine whether the results are too similar to determine the better language construct. The p-values for these results can be seen in Section A.2.3. Using the formula for percentage difference we find that there is a 0,67% difference between Package Energy measurements. Given this small difference, the change in what language constructs consume the least energy, and the results of the Kolmogorov-Smirnov tests, we cannot conclude which construct is better regarding energy consumption. We investigate why the difference in consumed energy is so small and why it varies which construct is the better one in Section 4.2.2

To determine the plausibility of these results we now perform our sanity check.

$$31,67 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 501.331,32\mu\text{J} \tag{4.5}$$

Equation 4.5 shows our estimated energy consumption for explicit casting. This value and the one recorded for the `as` benchmark have a difference of 17,59%, which is a plausible difference in accordance with our procedure presented in Section 4.1.

Findings

The results that are shown throughout this section do not help us determine what type of casting should be done, both in the case where we are looking at numeric conversion and conversion to a derived class.

When looking at our casting with derived classes, there is one major concern beyond the energy consumption of our options. As discussed in Section 2.1, there are differences in how explicit casting and the `as` keyword behave if casting fails.

In summary, the results are as follows:

- Numeric casting using implicit or explicit cast is inconsistent and the better construct varies from run to run.
- Derived/reference casting using either `as` or explicit cast suggests that `as` is the better option, however, results have a very small difference in energy consumption, making results inconclusive.

4.1.3 Parameter Mechanisms

In this section, the parameter mechanism benchmarks are presented alongside their results. This is the `Parameter Mechanisms` group presented in Chapter 2. This group is split into four subgroups. One group presents the results of using parameter mechanisms when the parameter passed is not modified, a group that looks at modifying passed reference-type parameters, a group that looks at modifying passed value-type parameters, and the last group which looks at how these parameter mechanisms can be used to return values.

Non-Modifying

The parameter mechanisms we create benchmarks for in this subgroup are:

- `in`
- `ref`

This group is constructed as `in` does not allow for the passed argument to be modified, we must test it against something that allows for the same behavior. Therefore, `ref` is utilized, as while it does allow the passed argument to be modified, the behavior of `in` can be replicated.

```
1 public class ParameterMechanismBenchmarks {
2     ...
3     [Benchmark("ParamMech", "Tests using the 'in' parameter
  ↪  modifier")]
4     public static ulong In() {
5             ulong result = 0;
6             for (ulong i = 0; i < LoopIterations; i++) {
7                     result = result + InArg(result) + i;
8             }
9             return result;
```

```
10      }
11      ...
12      private static ulong InArg(in ulong number){
13              return number;
14      }
15      ...
16  }
```

Listing 10: The benchmark for the `in` parameter modifier, which tests the effects of passing an unmodifiable parameter to a method.

In Listing 10 we can see the `in` benchmark in this group. In line 7, we can see the call to the `InArg` method, which is the method that employs the `in` parameter modifier. This method takes the value of `result` and returns it as seen at the bottom of the listing. This value is then added to `result` alongside `i`, which is used to make sure that the benchmark is not compiled away.

The benchmark for `ref`, is almost identical, except for the method called and how that method looks. The call is instead `RefArg(ref result)`, using the `ref` keyword. The called method also changes signature to `RefArg(ref ulong number)`, but otherwise remains identical.

Benchmark results

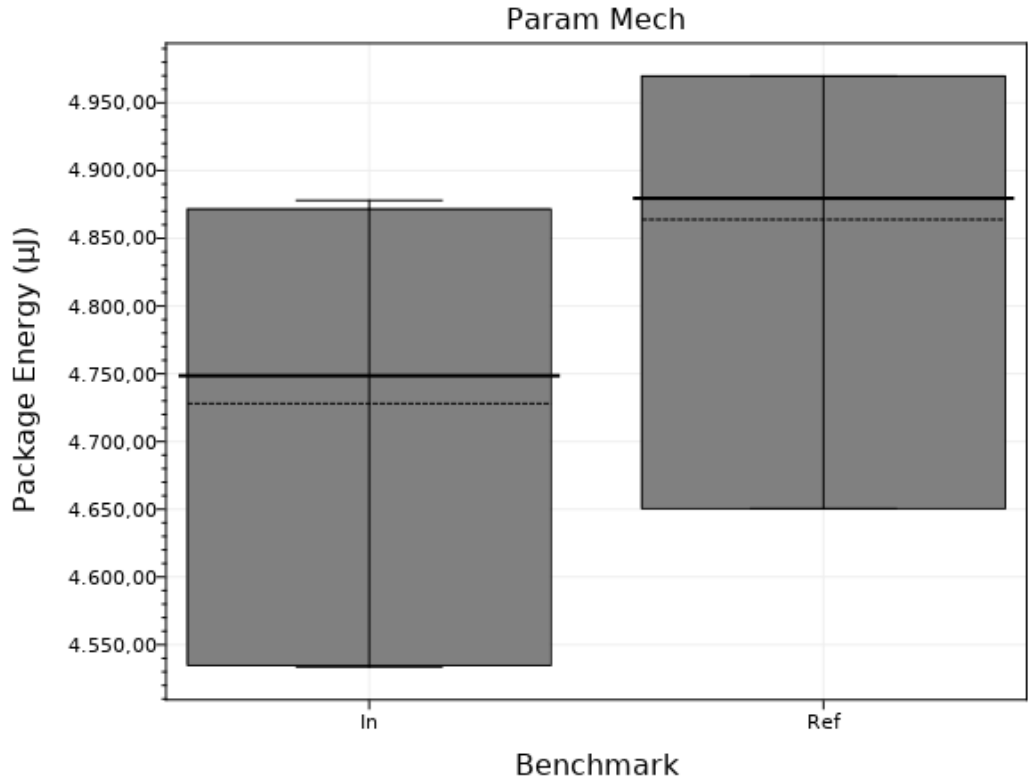

Figure 4.5: Boxplot showing Package Energy efficiency of different Param Mech Mod types. The y-axis starts at 4.510 μJ and ends at 4.990 μJ.

In Figure 4.5 we see the energy consumption of the two benchmarks, where a parameter is passed without modifying its value.

The plot shows how using `in` results in a lower energy consumption than using `ref`. Over the course of 10 runs, `in` has shown to be the better option in seven cases. The results of the 10 runs alongside the variance exhibited is seen in Section A.5, Table A.76. From our Kolmogorov-Smirnov tests, we have found that these language constructs are seemingly from the same probability distribution, this is seen in Section A.8.3, Table A.116. This varying nature is analyzed in Section 4.2.3.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Ref	0,489602	4.860,80	274,65
In	0,489601	4.727,54	274,72

Table 4.5: Table showing the elapsed time and energy measurement for each Param Mech benchmark.

In Table 4.5 we can see the numerical values for the results of our benchmarks. For a look at the p-values of these results see Section A.2.4. As the result are quite close to one another we check the percentage difference. We find a difference of 2,78% between the two results, which is not a level of difference large enough that we would consider `in` to be the more energy efficient language construct, because of the approximately 10% variance the benchmarks exhibit across their runs and the fact that our Kolmogorov-Smirnov tests suggest that they are part of the same probability distributions.

We now perform our sanity check of the plausibility of these results.

$$0,489602 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 7.750,40 \mu\text{J} \tag{4.6}$$

In Equation 4.6 we see our rough estimate for expected energy consumption for the `Ref` benchmark. Compared to the value recorded from our benchmark there is a percentage difference of 45,83%. Which falls inside our 100% cutoff for plausibility.

Modifying reference-type

In the group of constructs used for parameters which are modified we have:

- `ref`
- `out`

The reason that these language constructs are split into this group is that `out` requires a passed parameter to have a value assigned before the method terminates, and `in` cannot replicate this behavior, however `ref` can. The presentation of these benchmarks is done in the same way as the "Non-Modifying" subgroup. We present one benchmark and explain the changes made to test the second language construct.

```
1  public class ParameterMechanismBenchmarks {
2      ...
3      [Benchmark("ParamMechMod", "Tests using the 'ref' parameter
   ↪  modifier to modify value")]
4      public static ulong RefMod() {
5              ulong result = 0;
6              for (ulong i = 0; i < LoopIterations; i++) {
7                      result = result + RefModArg(ref result, i);
8              }
9              return result;
10     }
11     ...
12     private static ulong RefModArg(ref ulong number, ulong i){
13             number = i;
14             return number;
15     }
16     ...
17 }
```

Listing 11: The benchmark for the `ref` parameter modifier, which tests the effects of passing a parameter to a method and modifying it.

Listing 11 shows how the benchmark for `ref` looks in this subgroup. We see in line 7 the call to the `RefModArg`, which is where the `ref` modifier is used. Once more `ref` is used when calling the method as this is a requirement for methods that use `ref`. The method takes the `result` variable and modifies it by setting it to the value of `i` and adding the returned value to `result`. Using `i` we ensure the benchmark is not compiled away.

The other benchmark in this subgroup, `out`, changes the method called and how it is called. The call is now `OutArg(out result, i)` and the method changes signature to `OutArg(out ulong number, ulong i)`.

Benchmark results

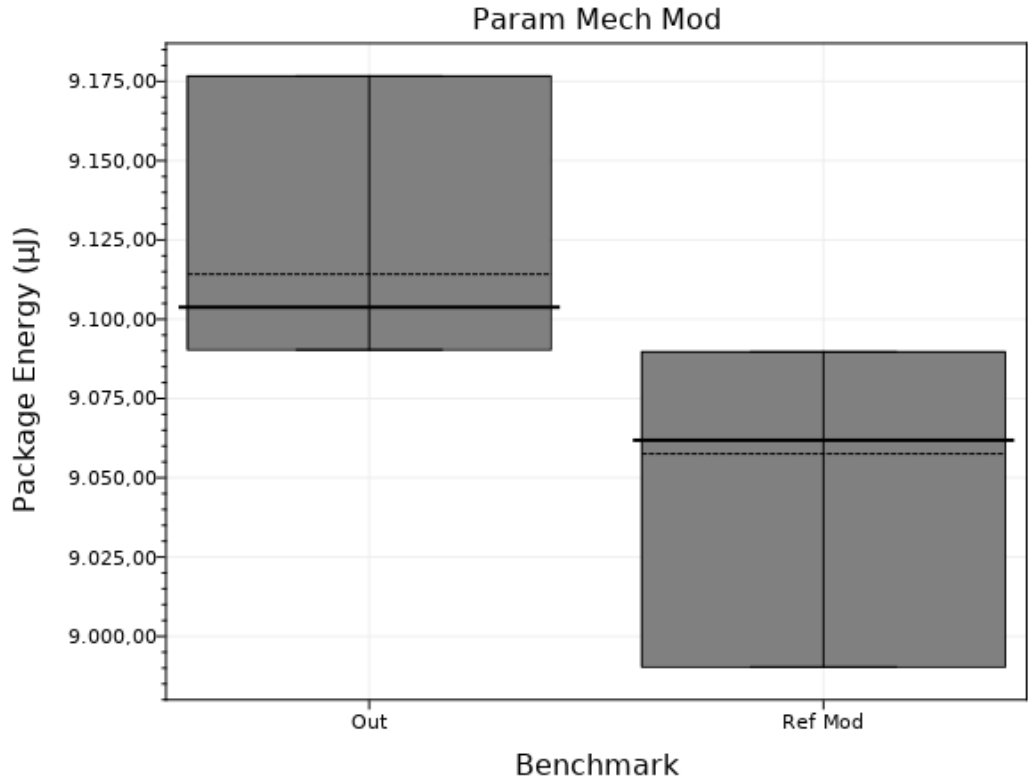

Figure 4.6: Boxplot showing Package Energy efficiency of different Param Mech Mod types. The y-axis starts at 8.980 μJ and ends at 9.185 μJ.

In Figure 4.6 we can see how our benchmark suggests that when passing an argument that should be modified inside the method the better option is using `ref`. These benchmarks have been run 10 times, and we have found that `ref` was better regarding energy consumption in 7 runs and `out` in 3 of the runs. The results of all 10 runs alongside the variance exhibited is seen in Section A.5, Table A.78. In Section A.8.3, Table A.117 the results of our Kolmogorov-Smirnov tests show us that the two language construct are probably from the same probability distribution.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Ref Mod	0,897	9.059,25	503,44
Out	0,899	9.115,43	504,47

Table 4.6: Table showing the elapsed time and energy measurement for each Param Mech Mod.

Table 4.6 shows the numerical values of the benchmark results. In Section A.2.5 the p-values of these results are seen. We now check the difference between the results for the two benchmarks, to determine if they are significantly different enough to be sure that the better performance is not due to small fluctuations between runs. We find a percentage difference of 0,62% between the Package Energy values measured, which is too small of a difference, for us to conclude whether one of these language constructs is more efficient than the other, especially given the variance of approximately 14% for both benchmarks and the fact that our Kolmogorov-Smirnov tests show that they are probably part of the same probability distribution.

We perform our sanity check to see if the results are plausible.

$$0,897 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 14.200,70\mu\text{J} \tag{4.7}$$

The rough estimate for the `Ref Mod` benchmark can be seen in Equation 4.7. The difference between what was measured, and the rough estimate is 44,21%, which we consider plausible, given our procedure for sanity checks and its assumptions.

Modifying value-type

In the group of constructs used for value-type parameters which are modified and returned we have:

- `ref`
- `"normal"`

As `ref` allows us to modify a parameter and use these changes outside of the method which changed, we need to compare it to equivalent functionality. Because neither `in` or `out` allows for this behavior we utilize `return`.A single benchmark is presented, and the changes required for refactoring them into the other benchmarks are explained.

```
public class ParameterMechanismBenchmarks {
    ...
    [Benchmark("ParamMechReturnStruct", "Tests using 'ref' to
    ↪ return a value")]
    public static ulong RefLargeStruct() {
            BigHelperStruct obj = new BigHelperStruct();
            for (ulong i = 0; i < LoopIterations; i++) {
                    ReturnModStructRef(ref obj, ref i);
            }
            return obj.Number;
    }
    ...
    private static void ReturnModStructRef(ref BigHelperStruct
    ↪ obj, ref ulong i) {
            obj.Number += i;
    }
    ...
}
```

Listing 12: The benchmark for the `ref` parameter modifier, which tests the effects of passing a parameter to a method and modifying it.

Listing 12 shows the benchmark for `ref` in this subgroup. Here, both parameters passed to `ReturnModStructRef` are passed using the `ref` keyword. There exists benchmarks for both `BigHelperStruct` as well as `SmallHelperStruct`, where the former is a struct which contains a `ulong` called `Number` as well as a string array with 100.000 strings, where string `i` is `i.ToString()`. The latter, `SmallHelperStruct`, only contains the `ulong Number`.

The difference in the "normal" construct is that instead of passing parameters using `ref`, the parameters are simply passed as `obj = ReturnModStruct(obj, i)`, and since `obj` is now passed by value, it has to be returned from the method and then assigned.

Benchmark results

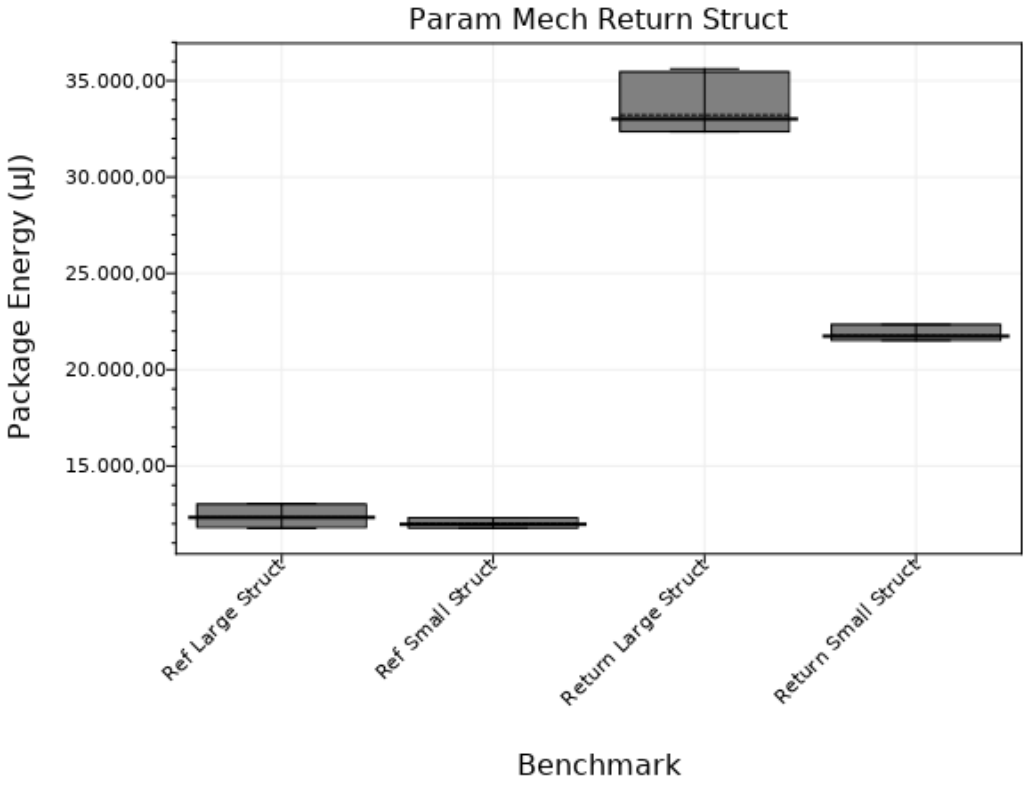

Figure 4.7: Boxplot Package Energy efficiency of different Param Mech Struct Mod types. The y-axis start at 11.000 μJ and ends at 37.000 μJ.

In Figure 4.7 we see that using `ref` when passing a value-type uses less energy than passing it normally and returning it afterwards.

Notably, the larger struct does not have a size-proportional influence on the energy consumption, as the energy consumption is $\approx$ 50% higher compared to the small struct for `return`, while the size is many times larger.We have found `ref` to be more energy-efficient across 10 runs. Across these runs, we have observed that there is a 50/50 split between whether `ref` with a small struct or a large struct has been measured to consume the least energy. This variance is not unexpected as the changes made to the structs are the same, and the struct is constructed once per iteration of a benchmark, meaning it has a negligible impact on the energy consumption. For a look at all 10 runs alongside the variance exhibited see Table A.80 in Section A.5. Looking at the results of our Kolmogorov-Smirnov tests in

Table A.118, Section A.8.3, we can see that all benchmarks are part of different probability distributions other than `Ref Large Struct` and `Ref Small Struct`. This is not unexpected as the only difference is the size of the struct, and as the struct is changed using `ref`, making it pass by reference the size does not affect consumption as much as the rest of the benchmark.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Ref Small Struct	1,31	12.012,93	727,38
Ref Large Struct	1,32	12.327,19	740,48
Return Small Struct	2,47	21.769,99	1.374,28
Return Large Struct	3,66	33.236,15	2.050,46

Table 4.7: Table showing the elapsed time and energy measurement for each Param Mech Return Struct.

Table 4.7 shows the numeric results from the benchmarks, and the p-values can be seen in Section A.2.6. We now check the percentage difference between the two `Small Struct` benchmarks, and afterward the difference between the `Large Struct` benchmarks. Comparing `Ref Small Struct` and `Return Small Struct` we find a percentage difference of 57,76%, thus the small structs we utilize there is a significant difference between using `ref` and returning normally. This conclusion is also based on the variance seen with the two benchmarks, where `Ref Small Struct` has a variance of 14,73% and `Return Small Struct` a variance of 10,57%. This is further supported by our Kolmogorov-Smirnov tests finding that they are part of different probability distributions.

This behavior is observed once more when we compare their `Large Struct` counterparts, where we find a difference of 91,78%. With the two benchmarks having a variance of 10,13% for `Ref Large Struct` and 11,03% for `Return Large Struct` there is no potential for overlap in the consumed energy between the two language constructs. This conclusion is also supported by the results of the Kolmogorov-Smirnov tests, which determine that they are part of different probability distributions. We can thereby conclude that `ref` is the better approach for modifying and returning value-types. Another observation made here is the difference between `Ref Small Struct` and `Ref Large Struct`, which is 2,58%, showing us that the size of the object potentially influences the power draw, however, we cannot say for certain as this difference might have been caused by the variance the benchmarks exhibit.

We perform our sanity check to see if the results are plausible, in this case, we look at the `Ref Small Struct` and how well the recorded energy consumption matches our rough estimate.

$$1,31 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 20.700,08\mu\text{J} \tag{4.8}$$

The rough estimate for the `Ref Small Struct` is seen in Equation 4.8. We now use this value together with the recorded result to find the percentage difference. The percentage difference is 53,11% which we consider plausible given the assumptions presented in the procedure for sanity checks, in Section 4.1.

Returning

In this group we have three language constructs that are used to construct benchmarks, these are:

- `ref`
- `out`
- `return`

In Chapter 2 we explained how `ref` and `out` worked, we saw from this that they can be used as a way of returning values from a method, therefore we compare this functionality from the constructs alongside a comparison of how `return` affects energy consumption when used for the same. We present one of the benchmarks for `ref` and `out` as they only differ in the use of a single keyword, while `return` has its own explanation.

```
1  public class ParameterMechanismBenchmarks {
2      ...
3      [Benchmark("ParamMechReturn", "Tests using the 'ref'
   ↪   parameter modifier to return a value")]
4      public static ulong RefReturn() {
5              ulong result = 0;
6              for (ulong i = 0; i < LoopIterations; i++) {
7                      ReturnRefHelp(ref result, i + result);
8                      result = result + i;
9              }
10             return result;
11     }
```

```
12      ...
13      private static void ReturnRefHelp(ref ulong number, ulong
        ↪  index) {
14              number = index + index;
15      }
16      ...
17  }
```

Listing 13: The benchmark for the `ref` parameter modifier, which tests the effects of using it to modify a parameter in a method and using the changed value, as one would use the value from a return statement.

In Listing 13 we can see how `ref` is used to mimic the behavior of `return`. This is done through the call to the `ReturnRefhelp` method. We here modify the value of `result` to be twice that of the argument that is passed.

The benchmark for `out` is identical except for the method call and its signature. The call is changed to `ReturnOutHelp(out result, i + result)`, which also reflects the changed signature of the method.

These benchmarks are then compared against `return`, to determine their comparative performance. The `return` benchmarks follow much of the same structure, however, the method that is called is no longer void, as values are not passed by reference, and any changes are discarded once the enclosing scope is exited.

```
1   public class ParameterMechanismBenchmarks {
2       ...
3       [Benchmark("ParamMechReturn", "Tests using 'return' to
    ↪   return a value")]
4       public static ulong Return() {
5               ulong result = 0;
6               for (ulong i = 0; i < LoopIterations; i++) {
7                       result = ReturnHelp(result, i + result);
8                       result = result + i;
9               }
10              return result;
11      }
12      ...
13      private static ulong ReturnHelp(ulong number, ulong index) {
```

```
14              number = index + index;
15              return number;
16      }
17      ...
18  }
```

Listing 14: The benchmark for `return`, which tests the effects of returning a value from a method.

In Listing 14 we can see what `ref` and `out` are compared to. It shows how `result` is assigned to the value returned by `ReturnHelp`, instead of the pass-by-reference approach used by the other benchmarks. This is one variation of how return can be used to return a value. Our second benchmark for `return`, changes this approach by using a method which only contains `return index + index;`.

Benchmark results

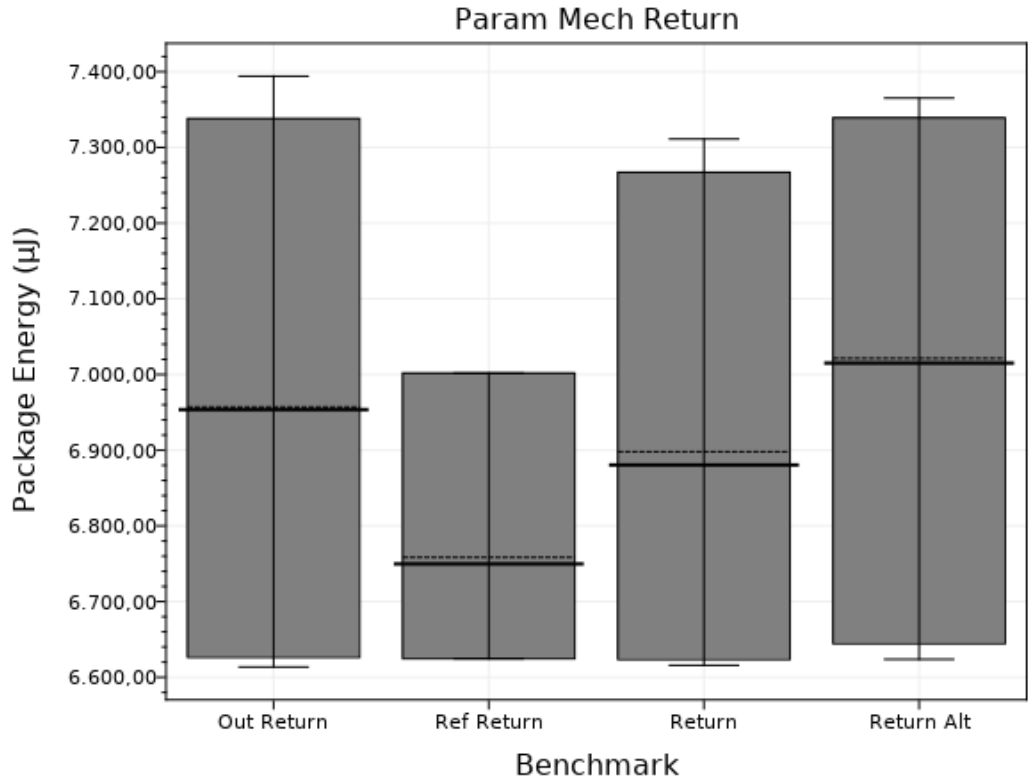

Figure 4.8: Boxplot showing package energy efficiency of different ways of returning a value. The y-axis spans from 6.580 μJ to 7.420 μJ.

In Figure 4.8 we can see the energy consumption of the four benchmarks used to alter/return a value from a method.

In the plot, we see that all the results can vary by a substantial amount. It shows how on average `ref` consumes the lowest amount of energy, outperforming the other benchmarks. Over 10 separate runs however these results do not remain the same. We have observed how `out` was the best in regard to performance in one run, `ref` four runs, regular `return` two runs, and the alternative approach in three runs. For a look at the results of the 10 runs alongside their variance see Table A.81 in Section A.5. When we look at the Kolmogorov-Smirnov tests in Section A.8.3, Table A.119 we see that none of the language constructs seem to be from a different probability distribution. The changing nature of the benchmarks is explored in Section 4.2.3.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Ref Return	0,734131	6.754,60	411,877
Return	0,734141	6.895,08	411,896
Return Alt	0,734137	7.018,57	411,922
Out Return	0,734135	6.956,92	411,921

Table 4.8: Table showing the elapsed time and energy measurement for each Param Mech Return.

Table 4.8 show the numerical values of the results from running our benchmarks. The p-values for the results can be seen in Section A.2.7. We check the two benchmarks with the lowest energy consumption against each other to find what the lowest percentage difference is, as it would then hold that if this difference is significant based on their variance, all others are too. We find that there is a difference of 2,06%, which means, coupled with the benchmarks variances which are approximately 10%, that we cannot determine whether one of these language constructs has better energy efficiency. Based on this we also check the difference between the highest and lowest result, we find a difference of 3,83%, which is too small of a difference to determine whether one of the language constructs is more energy efficient. These results are further supported by the fact that all results seem to be from the same probability distribution, according to the Kolmogorov-Smirnov tests. Before moving on to the findings of this section we perform our sanity check for determining the plausibility of the collected results.

$$0,734131 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 11.621,29\mu\text{J} \tag{4.9}$$

The rough estimate for returning a value using `ref` is seen in Equation 4.7. The difference between what was measured for the `Ref Return` and the estimate is 52,97%, which is plausible, given the assumptions for the procedure presented in Section 4.1.

Findings

We do not consider our results to be a solid basis for determining that using `in` is the better option when using parameter mechanisms for methods that do not modify the argument passed to the method, because the better construct varies between runs.

We cannot conclude that using `ref` consumed less energy for methods that alter the arguments passed, as the difference between energy consumption is below the specified threshold of 2%. One thing to add here is that there might be a difference in other use cases, as `ref` requires initialization, whereas `out` does not which can add to the energy consumption, however, this has not influenced our results as both were used with initialized variables.

Returning values using these different methods is too inconsistent to conclude which language constructs consume the least energy.

In summary:

- Passing arguments to methods that do not modify the passed arguments seems to be more energy efficient when using `in` compared to `ref`, however, results are inconsistent and vary between runs.
- Passing an argument to methods that modify the arguments is observed to consume less energy when using `ref`, however difference in energy consumption is too low to conclusively say that `ref` is better than `out`.
- Passing value-type parameters using `ref` is more energy-efficient than passing by value, however, the size of the passed argument does not seem to have a proportional influence on the power consumption for passing by value.
- The energy consumption of different methods of returning values is inconsistent, and no language construct which is better than others could be determined.

4.1.4 Lambda

This section looks at different approaches to using lambda expressions. While we look at one language construct, we have several ways to utilize it, which are presented in Chapter 2. Based on this we create the following benchmarks:

- `Lambda`
- `LambdaClosure`
- `LambdaParam`
- `LambdaDelegate`
- `LambdaDelegateClosure`
- `LambdaAction`

These benchmarks all achieve the same but help give an overview of some of the many ways lambda expressions can be used and how the different methods affect energy consumption. As before we do not show all the benchmarks, but rather give examples of what would be changed to make it into another one of our benchmarks.

```
1  public class LambdaBenchmarks {
2      ...
3      [Benchmark("Lambda Expression", "Tests a lambda expression
   ↪   using Func<> with closure")]
4      public static ulong LambdaClosure() {
5              ulong result = 0;
6              Func<ulong> testFunc = () => result + 10;
7              for (ulong i = 0; i < LoopIterations; i++) {
8                      result = testFunc() + i;
9              }
10             return result;
11     }
12     ...
13 }
```

Listing 15: The benchmark tests using a lambda expression with closure.

In Listing 15 we see how we test `LambdaClosure`. This benchmark looks at how having `result`, the variable, inside of the lambda expressions affects the energy consumption effects of a lambda expression. This benchmark is a variation of our `Lambda` benchmark.

The `Lambda` benchmark is different in how the lambda expression does not make use of closure, that is it does not capture the value of `result` and looks like this `Func<ulong> testFunc = () => 10` instead. To have the same results the call of the lambda expression is also different, with it being `result = result + testFunc() + i`, as opposed to what is seen in line 8.

Using Listing 15 we also explain how the `LambdaParam` benchmark functions. Once more the structure remains nearly identical, with the changes being how the lambda expression is changed to take a parameter, changing its form to `Func<ulong,ulong> testFunc = x => 10 + x`. This changes the call inside the `for` loop, now `result` is used as a parameter for the call of the lambda expression, changing the call to `result = testFunc(result) + i`.

```
public class LambdaBenchmarks {
    ...
    private delegate ulong Test();
    ...
    [Benchmark("Lambda Expression", "Tests a lambda expression
    ↪ using Delegate and closure")]
    public static ulong LambdaDelegateClosure() {
            ulong result = 0;
            Test testFunc = () => 10 + result;
            for (ulong i = 0; i < LoopIterations; i++) {
                    result = testFunc() + i;
            }
            return result;
    }
    ...
}
```

Listing 16: The benchmark for testing using a lambda expression with closure through a `delegate`.

We see in Listing 16 one of our two `delegate` benchmarks. These benchmarks are nearly identical to what is seen in Listing 15 for `LambdaClosure` and its variation `Lambda`, only changing from using `Func<>` to using `delegate` and then calling this delegate.

```
public class LambdaBenchmarks {
    ...
    [Benchmark("Lambda Expression", "Tests a lambda expression
    ↪ using Action")]
    public static ulong LambdaAction() {
            ulong result = 0;
            Action<ulong> test = param => result = 10 + param;
            for (ulong i = 0; i < LoopIterations; i++) {
                    test(result + i);
            }
            return result;
    }
```

```
12      ...
13  }
```

Listing 17: The benchmark for using a lambda expression with `Action<>`.

As the last benchmark example, we have `LambdaAction`. In this benchmark, we use our lambda expression in conjunction with `Action<>`. The variable `result` must be modified in the lambda expression that makes up our `Action<>`, the reason for this is that `Action<>` cannot return a value [48].

Benchmark results

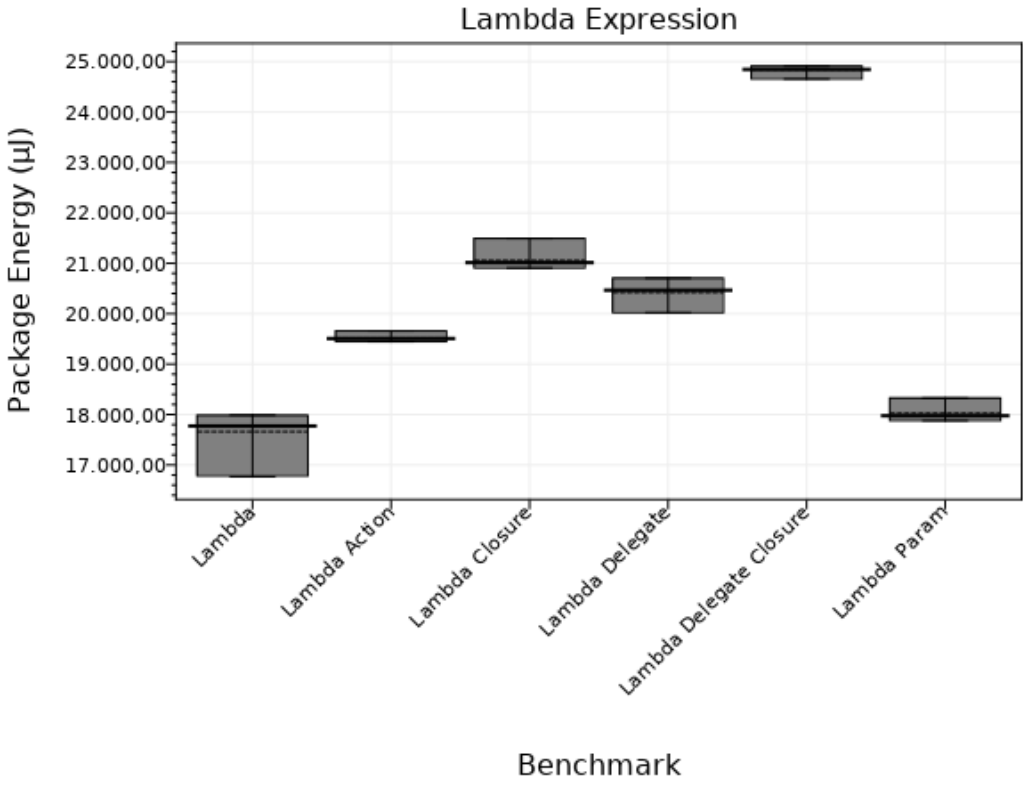

Figure 4.9: Boxplot showing Package Energy efficiency of different ways of using lambdas. The y-axis spans from 16.400 μJ to 25.200 μJ.

Figure 4.9 shows the energy consumption of our different lambda benchmarks.

We can see how using `Func<>` in our `Lambda` benchmark has the lowest energy consumption. We can also see how using a `closure` variation of a benchmark leads to higher energy consumption than otherwise, with the behavior exhibited by `Lambda Closure` and `Lambda Delegate Closure` compared to `Lambda` and `Lambda Delegate`. However, what is presented here is misleading. The benchmarks were run 10 times and the results were observed to vary from run to run. An example of this is how the `Lambda` benchmark was observed to consume the least energy in 6 runs, but second most in the other 4 runs. For a look of the results across 10 runs and the variance of each benchmark see Table A.83 in Section A.5. The results of the Kolmogorov-Smirnov tests in Table A.120, Section A.8.4, show that in a majority of the cases benchmarks that use a form of `delegate` are different from other constructs. We have that `Lambda` is from a different probability distribution than everything but `Lambda Param` and Lambda Delegate Closure is different to everything else. These results suggest that using closure or delegates has some effect that makes them distinct from other options. The cause behind this behavior is explored in Section 4.2.4.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Lambda Delegate Closure	2,036207	24.837,10	1.142,96
Lambda	1,468533	17.660,98	824,03
Lambda Action	1,468894	19.516,89	824,80
Lambda Delegate	1,712831	20.405,48	960,85
Lambda Closure	1,712809	21.063,91	961,64
Lambda Param	1,468307	18.018,34	824,48

Table 4.9: Table showing the elapsed time and energy measurement for each Lambda Expression.

The numerical values for the benchmarks are seen in Table 4.9. In Section A.2.8 the p-values for the results can be seen. First, we check the two benchmarks with the lowest energy consumption against each other to determine the difference. We find that the percentage difference between `Lambda` and `Lambda Param` is 2%, this difference coupled with the variance of the benchmarks, which is 24,44% for `Lambda` and 13,16% for `Lambda Param`, means that we cannot determine for these two whether one is more energy-efficient, which is further supported by the Kolmogorov-Smirnov tests showing that they are potentially from the same probability distributions. We also check the other end of the spectrum and look at `Lambda Closure` compared to `Lambda Delegate Closure`. With these, we

find a difference of 16,44%, with a variance of 6,82% and 25,48% respectively. This shows us that while there can be cases where `Lambda Delegate Closure` is more energy-efficient than `Lambda Closure`, this happens less often than for `Lambda` and `Lambda Param`, and in general not using a delegate will result in lower energy consumption, which is supported by how delegates are generally from different probability distributions according to the Kolmogorov-Smirnov tests. To add to this, we also look at the difference between `Lambda` and `Lambda Closure`. We find a difference between the two of 17,57% which suggests that `Lambda` consumes less energy in general. We can then add to this by looking at our Kolmogorov-Smirnov tests which show that they likely are from different probability distributions, supporting our conclusion that `Lambda` consumes less energy. However, we have from our variance that `Lambda Closure` is more stable in regards to energy consumption. Plausibility is once more checked by performing our sanity check for the `Lambda` benchmark.

$$1,468533 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 23.246,88 \mu\text{J} \tag{4.10}$$

We use the value seen in Equation 4.10 and check the percentage difference between this and the results recorded from the `Lambda` benchmark. Doing this we find a difference of 27,31%, which we deem as plausible under the assumptions presented for our procedure in Section 4.1.

Findings

While the information presented in this section does not give us a clear picture of what language construct consumes the least energy, the observation made suggests that using closure and/or delegates consumes more energy than not.

- Energy consumption of lambda expression varies between runs, leaving no clear language construct as the most energy-efficient.
- Lambda expressions that use closure or delegates seem to use more energy than their non-closure counterpart, however, they are less stable regarding energy consumption.

4.2 Result Analysis

The approach for result analysis builds on the one presented in [3, p. 111], and is as follows:

- Look at differences between benchmarks and see if any of the differences here explain differing energy consumption, keeping in mind that the difference that is looked at is not the language constructs under test,
- check if the language construct is from a library, and if that is the case the implementation and accompanying documentation is checked, otherwise, we look at the Microsoft documentation for the language constructs,
- look at the relevant parts of the Intermediate Language (IL) code, which is the IL that is different or expected to be, to find differences between the benchmarks, then see if these differences explain different energy consumption, and
- look at the assembly code and see if differences here explain the differing energy consumption between language constructs.

As mentioned in Section 3.2 we extend the analysis process to more consistently determine the reason behind differences. We analyze results showing that language constructs have the same effect, this helps to determine why there are no differences in energy consumption of two language constructs.

4.2.1 Concurrency

We now look into why there are differences between the energy consumption of the language constructs in this group.

Overhead with small workloads

First, we look into where the overhead comes from when using `Parallel.For` or the `Task` library with small workloads, meaning that `AmountOfWork` is 1. This is compared to using manual threading.

Looking at the benchmarks, they are constructed differently, as the constructs function differently, and use different constructs for summing the

partial results. The benchmarks using manual threading use X intermediate variables, where X is the number of threads used in the given benchmark. These intermediate variables are then summed sequentially after all threads are joined.

In the `Parallel.For` benchmarks, there is also a thread-local variable for each thread, and per default, the amount of threads is the number of cores available.

The `ManualThread4` benchmark and the default `Parallel.For` thus use the same amount of intermediate result variables. Since the same amount of intermediate results is used it does not explain the large gap in energy consumption between the two.

When using the `Task` library, a `Task<ulong>` object is created for each loop iteration, and this object contains the return value for the given task. Looking in the results file containing the information for the benchmark using the `Task` library with 1 `AmountOfWork`, there are 1.048.576 loop iterations. This means that significantly more intermediate objects are used when using the `Task` library the way that our benchmarks do, compared to `Parallel.For` and `ManualThread4`.

Looking at the benchmarks again, we see that the `Parallel.For` benchmarks utilizes `Interlocked.Add` to atomically add the partial result to the result, where our manual thread benchmarks use an array that stores each partial result, which is then later summed. As the function is only called four times when using four threads it is unlikely to have a significant impact, however, to be thorough we create a benchmark variation of `ManualThread4` that also uses the `Interlocked.Add` instead of storing the results in an array. Running this benchmark 10 times and comparing it to the original with a Kolmogorov-Smirnov test we get a p-value of 0,42, which means there is a chance that the two distributions are the same. Therefore we can not conclude that the use of `Interlocked.Add` is the cause of increased energy consumption.

To explain the differences, we look at the documentation for the different constructs. The documentation for `Parallel.For` [25] does not provide any specifics into how the library works, and as such, we look at the library code itself.

We find language constructs used in `Parallel.For`, implement them in the `ManualThread4` benchmarks, and see if the constructs make the `ManualThread4` benchmark less efficient and thus comparable with the `Parallel.For` benchmark. If this is the case, the language constructs used by `Parallel.For` is the reason for worse energy efficiency.

In [49] we see the implementation of the `Parallel` library and the comment at lines 5-7 describe that this construct internally uses the `Task` parallel library, giving us the first hint to why it is slower, as our benchmark using `Task` with 1 `AmountOfWork` is significantly less power efficient than `Parallel.For`.

On line 1.359 of the library a `RangeWorker` function called `FindNewWork` is called. Inspecting both classes, it is apparent that small slices of the loop are executed at a time by the different threads used by the `Parallel.For`. This slice is initially one but is multiplied by two in each iteration, up to a maximum value of 16. This means overheads occur each 16 loop iterations, by calling `FindNewWork`.

To test whether the overhead from `FindNewWork` is responsible, we manually use the `RangeWorker` in our `ManualThread4` benchmark to get loop slices.

```
1  public class Concurrency1 {
2      ...
3      public ulong ManualThread4RangeManager() {
4          ...
5          ulong res = 0;
6          RangeManager rangeManager = new RangeManager(0,
           ↪  (long)LoopIterations, 1, (int)numThreads);
7          for (ulong j = 0; j < numThreads; j++) {
8              tArr[j] = new Thread(() => {
9                  ulong temp = 0;
10                 RangeWorker currentWorker = default;
11                 if (!currentWorker.IsInitialized) {
12                     currentWorker =
                       ↪  rangeManager.RegisterNewWorker();
13                 }
14                 long nFromInclusiveLocal;
15                 long nToExclusiveLocal;
16                 if (currentWorker.FindNewWork(out
                   ↪  nFromInclusiveLocal, out nToExclusiveLocal)
                   ↪  == false) {
17                     return; // no need to run
18                 }
19                 do {
```

```
20                      for (long i = nFromInclusiveLocal; i <
                         ↪  nToExclusiveLocal; i++) {
21                          temp += SharedResources.DoSomeWork(1);
22                      }
23                  } while (currentWorker.FindNewWork(out
                     ↪  nFromInclusiveLocal, out
                     ↪  nToExclusiveLocal));
24                  Interlocked.Add(ref res, temp);
25              });
26              tArr[j].Start();
27          }
28          ...
29      }
30      ...
31  }
```

Listing 18: The benchmark which tests using threads manually with `RangeWorker`.

In Listing 18 we see a rewritten `ManualThread4` benchmark that uses `RangeManager` from [50] to retrieve loop slices. The `RangeManager` implementation has to be copied to our namespace as the class is marked `internal`. `RangeManager` and `RangeWorker` are used in the same way as in [49].

Package Energy (μJ)	ManualThread4 RangeWorker
Run 1	21.691,37
Run 2	22.074,27
Run 3	21.541,63
Run 4	21.683,66
Run 5	21.746,48
Run 6	21.769,00
Run 7	21.889,53
Run 8	21.922,76
Run 9	22.067,79
Run 10	22.115,90
Mean	21.850,24
Percentage variance	2,60%

Table 4.10: Results of running the ManualThread using `RangeManager` benchmark 10 times with mean and percentage variance between highest and lowest recorded energy consumption.

While the average in Listing 18 is 11,97% higher than the original `ManualThread4` benchmark, seen in Table A.72 in Section A.5, the power consumption is still below half of the `Parallel.For` benchmark. This means that the `RangeManager` offers part of the explanation.

Looking further at [49] we see at line 1.403 the start of the loop that is executing the lambda given to the `Parallel.For`. We also see on lines 1.405 and 1.415 that two `if` checks occur in every loop slice before the lambda, which could be a source of overhead.

Package Energy (μJ)	ManualThread4 RangeManagerTwoIf
Run 1	23.400,09
Run 2	23.064,61
Run 3	23.126,20
Run 4	23.478,15
Run 5	23.058,79
Run 6	23.059,54
Run 7	23.134,52
Run 8	23.086,49
Run 9	23.160,53
Run 10	23.091,81
Mean	23.166,07
Percentage variance	1,79%

Table 4.11: Results of running the ManualThread benchmark using `RangeManager` and having two if statements 10 times with mean and percentage variance between highest and lowest recorded energy consumption.

We calculate that the energy consumption in Table 4.11 is 6,02% higher than in Table 4.10, and given the small variance, we find that the redundant `if` statements make a difference, but are still below half of the most energy efficient `Parallel.For` benchmark.

Next, we test the overhead that comes from calling a lambda instead of having the body directly in the benchmark. Therefore, we construct a benchmark where we add the use of lambda instead of the method body.

Package Energy (μJ)	ManualThread4 RangeManagerLambda
Run 1	38.326,99
Run 2	39.304,12
Run 3	38.475,11
Run 4	38.989,03
Run 5	40.854,00
Run 6	39.449,61
Run 7	39.228,98
Run 8	38.710,32
Run 9	38.363,94
Run 10	39.280,41
Mean	39.098,25
Percentage variance	6,19%

Table 4.12: Results of running the ManualThread benchmark using `RangeManager` having two if statements, and using lambda instead of method body 10 times with mean and percentage variance between highest and lowest recorded energy consumption.

In Table 4.12 adding the use of lambda increases the power consumption by 68,77% compared to Table 4.11. This brings us closer to the result of the power consumption of the most power-efficient `Parallel.For` benchmark.

Another possible source is the use of `TaskReplicator` [51] in the `Parallel.For` implementation to handle concurrency, which is using the `Task` library. Therefore we create a benchmark that uses `TaskReplicator` instead of our manual threading, to see how this contributes to energy consumption.

Package Energy (μJ)	TaskReplicator Manual
Run 1	47.269,82
Run 2	44.013,94
Run 3	44.807,41
Run 4	44.215,43
Run 5	43.882,32
Run 6	43.838,05
Run 7	43.852,76
Run 8	43.905,72
Run 9	44.049,62
Run 10	44.219,25
Mean	44.405,43
Percentage variance	7,26%

Table 4.13: Results of running the manual `TaskReplicator` benchmark 10 times with mean and percentage variance between highest and lowest recorded energy consumption.

In Table 4.13 we see the results of running our benchmark with `TaskReplicator` instead of manual threading. This increases the average energy consumption with 13,57% compared to Table 4.12, and thus brings us closer to the energy consumption of the `Parallel.For` benchmark. This tells us that this contributes to the higher energy consumption `Parallel.For` has in comparison to manual threading.

Seen throughout this section, the use of loop slicing, additional `if` statements, lambda, and `TaskManager` increase the energy consumption of `Parallel.For`. In the implementation of `Parallel.For` it is also needed to handle exceptions thrown in the loop, keep track of the loop state, and make it generic, all of which are unnecessary features in our naive implementation of using threads manually [49]. All of these factors result in the increased power consumption of `Parallel.For` when the work done per loop iteration is small.

ThreadPool overhead

Our benchmark using `ThreadPool` uses significantly more energy than manually using threads or `Parallel.For` throughout the different amounts of work. `ThreadPool` also starts consuming more energy than using the `Task` library for `AmountOfWork` above 10.000.

We start by looking for differences in the benchmarks. Looking at the `ThreadPool` benchmarks from Listing 6 and Listing 7 in Section 4.1.1, they use an array of optional `ulong`, where there is an array entry for each loop iteration.

We also see in Listing 7 that the benchmark busy waits using `Thread.Sleep(0);` until a value is present in the array cell, as there is no other way to wait for all tasks to be complete.

To test if the use of the results array is the cause of increased energy consumption, a variation is created which instead calls `Interlocked.Add`, and does the same busy-wait until there are no remaining tasks in the `ThreadPool`, which works because it is the only process adding tasks to it.

We hypothesize that this result has the same or worse power consumption for smaller workloads, as using `Interlocked.Add` may cause synchronization overhead if the operation is called simultaneously from multiple threads.

Package Energy (μJ)	ThreadPool1Interlocked
Run 1	3.045.150,57
Run 2	3.154.591,59
Run 3	3.279.986,79
Run 4	3.405.424,22
Run 5	3.049.132,67
Run 6	3.228.575,29
Run 7	3.090.600,04
Run 8	3.156.921,06
Run 9	3.282.489,92
Run 10	3.259.191,19
Mean	3.195.206,33
Percentage variance	10,58%

Table 4.14: Results of running the `ThreadPool1Interlocked` benchmark 10 times with mean and percentage variance between highest and lowest recorded energy consumption.

In Table 4.14 we see the results of using `Interlocked.Add` instead of storing the results in an array. Compared to the `ThreadPool1` benchmark the mean energy consumption is decreased by 10,97%. Because of the large variance in the benchmarks, a Kolmogorov-Smirnov test is conducted to determine if they are drawn from the same probability distribution. This results in a p-value of below 0,00, which means the distributions are different with a high probability. This means that the results are contradictory to our hypothesis.

To test if the busy-wait is the culprit, we modify the benchmark used in Table 4.14 to instead utilize a `CountdownEvent` [52], which is signaled after the `Interlocked.Add` operation is finished and contains a method called `Wait`, which waits until the countdown has reached 0.

We hypothesize the performance will be worse for small workloads, as there is a constant overhead of using a thread pool, and if the workload is small, the overhead constitutes a larger portion of the total work. In the case of larger workloads, the performance should be improved as it constitutes a smaller portion of the total work and results in the main process not busy-waiting.

Package Energy (μJ)	ThreadPool1Countdown
Run 1	2.915.167,44
Run 2	2.785.189,44
Run 3	2.758.928,09
Run 4	2.810.724,53
Run 5	3.092.053,92
Run 6	2.879.560,99
Run 7	3.178.323,03
Run 8	2.805.345,89
Run 9	3.076.481,47
Run 10	3.041.154,91
Mean	2.934.292,97
Percentage variance	13,20%

Table 4.15: Results of running the `ThreadPool1Countdown` benchmark 10 times with mean and percentage variance between highest and lowest recorded energy consumption.

The results of using the `CountdownEvent` are seen in Table 4.15, we have that on average 8,17% less energy is used than in Table 4.14. Performing a Kolmogorov-Smirnov test, which gives a p-value of 0,012, meaning that there is a high probability that the distributions are different. While more energy-efficient there is a significant difference in energy consumption compared to the `Parallel.For` and manual thread benchmarks.

To see if the higher energy consumption stems from overhead in the `ThreadPool` implementation, a trimmed down naive version of a thread-pool is created, which uses a `ConcurrentQueue` of `Actions`, and threads that dequeues and executes the `Actions`.

Package Energy (μJ)	ThreadPool1Naive
Run 1	2.481.242,66
Run 2	2.497.708,98
Run 3	2.513.147,89
Run 4	2.477.653,89
Run 5	2.493.912,66
Run 6	2.475.311,39
Run 7	2.500.366,33
Run 8	2.510.181,84
Run 9	2.482.837,44
Run 10	2.488.980,62
Mean	2.492.134,37
Percentage variance	1,51%

Table 4.16: Results of running the `ThreadPool1Naive` benchmark 10 times with mean and percentage variance between highest and lowest recorded energy consumption.

The results of running the naive implementation are shown in Table 4.16, and the mean value is 15,07% lower than Table 4.15. While the energy efficiency is improved, there is still a significant difference between `Parallel.For` and the manual benchmarks. To test if the increased amount of calls to `Interlocked.Add` are the cause, a specialized version of the naive thread pool is created, where each thread has a local subtotal variable, and `Interlocked.Add` is only called once per thread when no more tasks remain.

Package Energy (μJ)	ThreadPool1Specialized
Run 1	2.371.147,10
Run 2	2.317.340,10
Run 3	2.366.877,51
Run 4	2.339.735,97
Run 5	2.337.352,72
Run 6	2.325.815,54
Run 7	2.331.184,10
Run 8	2.343.275,14
Run 9	2.331.775,34
Run 10	2.329.444,31
Mean	2.339.394,78
Percentage variance	2,27%

Table 4.17: Results of running the `ThreadPool1Specialized` benchmark 10 times with mean and percentage variance between highest and lowest recorded energy consumption.

In Table 4.17 the results of running the specialized version of a thread pool are seen. It calls `Interlocked.Add` once per thread created, instead of once per loop iteration. The results are on average 6,13% below Table 4.16 and a Kolmogorov-Smirnov test gives a p-value of below 0,000, which means there is a low probability that results are from the same distribution.

This leaves the overhead of queuing the work, which is tested by having the sequential benchmark compared with a benchmark that calls a `WaitCallback` containing the work, which is the delegate used by `ThreadPool` when queuing work, and another benchmark which first queues the work in a `ConcurrentQueue` wrapped as a `WaitCallback`, then dequeue it and executes the work. This gives the overhead of executing a delegate and adding the work to a queue. However, the overhead of multiple threads trying to dequeue at the same time is not captured.

Package Energy (μJ)	NoQueue1	NoQueue1WaitCallback	ConcurrentQueue1
Run 1	21.211,62	39.726,11	343.711,43
Run 2	22.155,53	42.191,35	345.372,71
Run 3	21.191,37	42.265,79	345.272,75
Run 4	21.264,46	39.945,47	346.090,53
Run 5	21.401,32	41.961,57	342.914,45
Run 6	21.164,49	41.898,95	341.676,61
Run 7	20.877,46	39.004,17	344.046,13
Run 8	21.192,78	42.867,00	346.577,44
Run 9	22.112,33	42.092,76	348.999,91
Run 10	21.133,74	39.602,89	343.568,98
Mean	21.370,51	41.155,61	344.823,09
Percentage variance	5,77%	9,01%	2,10%

Table 4.18: Results of running the `NoQueue1`, `NoQueue1WaitCallback`, and `ConcurrentQueue1` benchmark 10 times with mean and percentage variance between highest and lowest recorded energy consumption.

The results of running the benchmarks with and without the use of a `ConcurrentQueue` are seen in Table 4.18. Using the `WaitCallback` delegate increases the energy consumption by 92,58% compared to the `NoQueue` benchmark, and the use of a concurrent queue increases the energy consumption by 1513,55% compared to `NoQueue`.

The overhead of using a `ConcurrentQueue` is significant, and as such a variation of the naive thread pool is created, where instead of having a `ConcurrentQueue` of work, the work is specified in a `WaitCallback`, and a variable which specifies the number of executions. This construct is no longer a thread pool as it can only run a single predefined workload, but that is necessary to avoid using a `ConcurrentQueue`.

Package Energy (μJ)	ThreadPool1NoQueue
Run 1	994.326,29
Run 2	1.000.118,32
Run 3	996.835,49
Run 4	993.478,59
Run 5	994.742,58
Run 6	996.841,69
Run 7	989.554,97
Run 8	988.034,93
Run 9	994.337,90
Run 10	999.277,15
Mean	994.754,79
Percentage variance	1,21%

Table 4.19: Results of running the `ThreadPool1NoQueue` benchmark 10 times with mean and percentage variance between highest and lowest recorded energy consumption.

In Table 4.19 we see the results of running the naive thread pool without any queues. This approach uses 72,28% less energy than the original `ThreadPool1` benchmark. While significantly less, it is still significantly worse than `ManualThread4` or `ParallelForDefault`.

In conclusion, when workloads are small, using a `ThreadPool` or `Task` based approach is inefficient, as each small workload is executed separately from the others, as opposed to i.e. `ManualThread4` where each thread runs one-fourth of the loop iterations, or in `Parallel.For` where the loop is split into loop slices. Furthermore, the use of a queue for storing the work also significantly worsens performance and energy consumption. Here we can see that `ThreadPool` has a different purpose than `Parallel.For`, as the former is meant to be able to execute different workloads while the latter is for doing a single workload many times.

However, when the workloads are larger, the overhead between the workloads becomes negligible, and the optimized version of `ThreadPool` using `CountdownEvent` is just as efficient as `ManualThread4` and `ParallelForDefault`. In our case, the overhead becomes negligible at about 10.000 `AmountOfWork`.

4.2.2 Casting

For *Casting*, we look into the reason why results vary between runs as was observed in Section 4.1.2.

Numeric Casting

To determine why we observe a difference between runs in Section 4.1.2, we look at the average consumed package energy across 10 runs.

In Section A.5, Table A.74 we have the results of running our benchmarks 10 times. We can see that the `Implicit` and `Explicit` benchmarks have a percentage variance of 17,46% and 7,01% respectively. This degree of variance is outside the average variance found in Section 3.3 and as such we look into the cause.

In an effort to eliminate this variance we run the benchmark 10 more times, using `GC.TryStartNoGCRegion` [53] and `GC.EndNoGCRegion` [54] around the benchmark execution in an effort to stop garbage collection.

Table A.75 in Section A.5 shows that turning garbage collection off has not changed the fact that there is variance in the results, in fact the variance has increased to 22,83% for `Implicit` and 13,49% for `Explicit`.

As we have not been capable of stabilizing the results, we cannot determine which, if any, of the two language constructs consumes less energy. Because of this, we use our analysis method, to instead explain the observed variance.

First, we look at the benchmarks themselves to see if there is a difference in these that might explain this variance. In Section 4.1.2, Listing 8, we can see the `Explicit` benchmark. The only difference between the `Explicit` and the `Implicit` benchmark is the use of explicit conversion instead of implicit conversion.

Given that this is the only difference between the benchmarks we look at the documentation for explicit conversion and implicit conversion. In the documentation [55] we find that all implicit conversions are classified as explicit conversions. This leads to the idea that the IL code is identical as implicit conversions are explicit conversions.

```
1 // result = result + i / result = result + (double)i;
2 IL_000f: ldloc.0
3 IL_0010: ldloc.1
4 IL_0011: conv.r.un
5 IL_0012: conv.r8
6 IL_0013: add
7 IL_0014: stloc.0
```

Listing 19: The `Implicit/Explicit` benchmarks important IL code.

The code presented in Listing 19 shows that there are no differences between the approaches, as expected. We conclude that the benchmarks, are identical and both actually perform an explicit conversion. The variance is not explained by this, however, we deem this irrelevant on the basis that using an explicit cast where an implicit conversion is possible is the same.

Derived Casting

We now look into why there is no difference between using `as` or explicit casting. In Section 4.1.2, Listing 9, we can see how the `As` benchmark functions, alongside an explanation of the differences between it and the `ReferenceExplicit` benchmark. The only difference between the two happens inside of their `for` loop, where `as` makes use of `result = (o as string) + i;` and `ReferenceExplicit` uses `result = (string)o + i;`. This difference does not explain the similarity between approaches.

Per the documentation [31] the `as` keyword is equivalent to `E is T ? (T)(E) : (T)null`, however the expression `E` is only evaluated once. Given that `E is T ? (T)(E) : (T)null` makes use of `(T)(E)`, which is a cast expression, and that it evaluates more than once, we have that `(T)E` evaluates once. From this, we have that there is no difference in the number of evaluations between the two constructs. In Section 2.1 we also established that the major difference between the `As` and `ReferenceExplicit` benchmarks were how the language constructs handled unsuccessful type conversion, with `as` returning `null` and explicit casting throwing an exception. Neither of these happens in our benchmarks and as such the two language constructs seem to be the same, as was observed in regard to energy consumption.

We do not consider this an acceptable explanation for why the results of these language constructs are so similar and therefore look at the IL code.

```
1 // result = (o as string) + i;
2 IL_0011: ldloc.1
3 IL_0012: isinst       [System.Private.CoreLib]System.String
4 IL_0017: ldloca.s     2
5 IL_0019: call         instance string
  ↪  [System.Private.CoreLib]System.UInt64::ToString()
6 IL_001e: call         string
  ↪  [System.Private.CoreLib]System.String::Concat(string,
  ↪  string)
7 IL_0023: stloc.0
```

Listing 20: The `As` benchmarks important IL code.

In Listing 20 we can see the relevant code for the `As` benchmark. The important detail here is the use of the `isinst` instruction.

```
1 // result = (string)o + i;
2 IL_0011: ldloc.1
3 IL_0012: castclass    [System.Private.CoreLib]System.String
4 IL_0017: ldloca.s     2
5 IL_0019: call         instance string
  ↪  [System.Private.CoreLib]System.UInt64::ToString()
6 IL_001e: call         string
  ↪  [System.Private.CoreLib]System.String::Concat(string,
  ↪  string)
7 IL_0023: stloc.0
```

Listing 21: The `ReferenceExplicit` benchmarks important IL code.

In Listing 21 the relevant code for the `ReferenceExplicit` benchmark is presented, with the important part being the use of the `castclass` instruction.

We observe that the two benchmarks have a difference in their IL, which suggests that they should have differing energy consumption. However

looking at the documentation for the `isinst` [56] and `castclass` [57] instructions the documentation for `isinst` states: *"Tests if an object reference is an instance of `class`, returning either a null reference or an instance of that class or interface."* [56]. This is done through three steps:

1. *"An object reference is pushed onto the stack."*
2. *"The object reference is popped from the stack and tested to see if it is an instance of the class passed in class."*
3. *"The result (either an object reference or a null reference) is pushed onto the stack."*

It is also stated that "If the class of the object on the top of the stack implements `class` (if `class` is an interface) or is a derived class of `class` (if `class` is a regular class) then it is cast to type `class` and the result is pushed on the stack, exactly as though castclass had been called." [56]. From this, we conclude that the behavior is identical to `castclass`, which is the only difference between the IL, in cases where conversion is possible. This explains why there is no difference between using `as` or explicit casting.

4.2.3 Parameter Mechanisms

We analyze our results to determine why there are differences between the energy consumption of the different language constructs.

Non-Modifying

To begin we analyze why there is a variance between the runs observed in Section 4.1.3.

Table A.76 in Section A.5 shows us the result of our 10 runs of the benchmarks, we can see that both of the benchmarks vary by approximately 10%. To determine why this happens we look at the documentation for `in` and `ref` [32, 34, 33]. We find that they cause arguments to be passed by reference, which leads us to the idea that the cause of variance is garbage collection. To test this hypothesis we run the benchmarks with garbage collection disabled.

We see in Section A.5, Table A.77 how turning the garbage collection off has reduced the variance from approximately 10% to approximately 4%. This partially confirms our hypothesis, that garbage collection is part of the cause of the variance. We can see that both before and after turning

off garbage collection they both exhibit approximately the same variance. Another thing to note is the variance between the language constructs. The percentage variance between the lowest result from `in` and the highest result from `ref` is 10,40%, while the variance is 10,31% with garbage collection on. These observations suggest that the two language constructs have similar if not identical implementations.

Looking at the code for the benchmarks Listing 10 in Section 4.1.3, we see how the `In` benchmark functions, alongside an explanation of the differences compared to the `Ref` benchmark. We see that the only difference is in the call of method `InArg(result)` and `Ref RefArg(ref result)`, which only differ in signature. This difference is expected as these are the differences we are testing.

According to the documentation [33, 34], the `in` parameter modifier *"... is like the `ref` or `out` keywords, except that `in` arguments cannot be modified by the called method."* [33, 34], another commonality is how the documentation states that they *"... makes the formal parameter an alias for the argument, which must be a variable."* [33, 34] This suggests that the two language constructs are quite similar.As there is no library implementation for these language constructs we look at the IL code.

```
1 // result = result + InArg(result) + i;
2 IL_0008: ldloc.0
3 IL_0009: ldloca.s     0
4 IL_000b: call         uint64
  ↪  CompilerGen/ParameterMechanismBenchmarks::InArg(uint64&)
5 IL_0010: add
6 IL_0011: ldloc.1
7 IL_0012: add
8 IL_0013: stloc.0
```

Listing 22: The `In` benchmarks important IL code.

The code in Listing 22 and the code for `Ref` is nearly identical, the difference between them is the called method

.

```
.method private hidebysig static
uint64 InArg (
    [in] uint64& number
) cil managed
{
    .param [1]
    .custom instance void
    ↪ [System.Private.CoreLib]System.Runtime.CompilerServices.
    ↪ IsReadOnlyAttribute::.ctor() = (
    01 00 00 00
    )
    .maxstack 8
    // return number;
    IL_0000: ldarg.0
    IL_0001: ldind.i8
    IL_0002: ret
}
```

Listing 23: The `InArg` methods IL code.

```
.method private hidebysig static
uint64 RefArg (
    uint64& number
) cil managed
{
    .maxstack 8
    // return number;
    IL_0000: ldarg.0
    IL_0001: ldind.i8
    IL_0002: ret
}
```

Listing 24: The `RefArg` methods IL code.

In Listing 23 and Listing 24 we see a difference, which is the call of the constructor of the `IsReadOnlyAttribute` class [58]. This marks the argument passed with the use of `in` as a read-only program element. This attribute is used by the compiler to track metadata. This suggests that there is a difference between them, however, this is not what we have observed, from our results, and as such this does not explain the similarities between the benchmarks.

```
ParameterMechanismBenchmarks+UInt64 In():
  0000: push   rdi
  0001: push   rsi
  0002: sub    rsp, 28h
  0006: xor    esi, esi
  0008: xor    edi, edi
  000A: mov    rcx, 7FFEA28CCE08h
  0014: mov    edx, 1
  0019: call
   ↪ CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE_DYNAMICCLASS
   ↪ 5D8CCB60
  001E: mov    rax, [rax+10h]
  0022: test   rax, rax
  0025: je     short 0035
  0027: add    rsi, rsi
  002A: add    rsi, rdi
  002D: inc    rdi
  0030: cmp    rax, rdi
  0033: ja     short 0027
  0035: mov    rax, rsi
  0038: add    rsp, 28h
  003C: pop    rsi
  003D: pop    rdi
  003E: ret
  ...
  ParameterMechanismBenchmarks+UInt64 InArg(UInt64&):
  0000: mov    rax, [rcx]
  0003: ret
```

Listing 25: The `In` benchmark translated to ASM code.

```
ParameterMechanismBenchmarks+UInt64 Ref():
  0000: push   rdi
  0001: push   rsi
  0002: sub    rsp, 28h
  0006: xor    esi, esi
  0008: xor    edi, edi
  000A: mov    rcx, 7FFEA28CCE08h
  0014: mov    edx, 1
  0019: call
   ↪ CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE_DYNAMICCLASS
   ↪ 5D8CCB00
  001E: mov    rax, [rax+10h]
  0022: test   rax, rax
  0025: je     short 0035
  0027: add    rsi, rsi
  002A: add    rsi, rdi
  002D: inc    rdi
  0030: cmp    rax, rdi
  0033: ja     short 0027
  0035: mov    rax, rsi
  0038: add    rsp, 28h
  003C: pop    rsi
  003D: pop    rdi
  003E: ret
  ...
  ParameterMechanismBenchmarks+UInt64 RefArg(UInt64&):
  0000: mov    rax, [rcx]
  0003: ret
```

Listing 26: The `Ref` benchmark translated to ASM code.

Looking at the assembly code in Listing 25 and Listing 26 we see that they are identical, which confirms our suspicion from earlier that the code run for the language constructs is identical when it has been compiled. This also means that the variance in the results is irrelevant on the basis that using `in` or `ref` for a method where no modifications happen is the same.

Modifying reference-type

We look into why there is a variance between the runs in Section 4.1.3. Table A.78 in Section A.5 shows how the two language construct have approximately the same variance around 14%. These language constructs cause arguments to be passed by reference [35, 34], which leads to us running them without garbage collection. In Table A.79 we see no tangible difference in variance compared to having garbage collection on, except for `Out` having 2% more variance. Thereby it is unlikely that garbage collection is the cause of the variance.

The variance between the lowest result from `ref` and the highest result from `out` is 14,42% and 13,60% the other way around with garbage collection on and for garbage collection off the variance between lowest result from `ref` and the highest result from `out` is 12,28% and 17,77% the other way around. Based on these observations we hypothesize that the two language constructs are more or less identical in their implementation.

To test whether the language constructs are identical in IL or assembly, we proceed with our analysis of the results. If we compare the code in Section 4.1.3, Listing 11 we get an example of how the `ref` benchmark works alongside an explanation of the `out` benchmark. We find here that the benchmarks are identical except for two different methods which are called, with the calls being `RefModArg(ref result, i)` and `OutArg(out result, i)`. The two methods do the same, with the only difference being the parameter mechanism used. These differences are as expected and give us no further insight into whether or not the benchmarks are the same.

The documentation [34, 35] states for both "... makes the formal parameter an alias for the argument, which must be a variable. In other words, any operation on the parameter is made on the argument.", they both also make so arguments are passed by reference. The documentation does not tell why these language constructs have the same energy impact, because even though they are similar and have the same behavior in large parts, there are still differences between them. The information gathered from the documentation suggests that depending on the use case the language constructs do not differ, and therefore are, essentially the same. The major difference lies in the requirements of their use. We have from this that as long as the variable passed to the parameter modifiers is initialized they should be the same. However, we look at the IL code to see if this holds once compiled.

```
// result = result + RefModArg(ref result, i);
IL_0008: ldloc.0
IL_0009: ldloca.s     0
IL_000b: ldloc.1
IL_000c: call         uint64
    ↪  CompilerGen/ParameterMechanismBenchmarks::RefModArg(uint64&,
    ↪  uint64)
IL_0011: add
IL_0012: stloc.0
```

Listing 27: The `RefMod` benchmarks important IL code.

The code in Listing 27 shows the IL code for both `Ref` and `Out` with the only difference between the two being the called method.

```
.method private hidebysig static
uint64 OutArg (
    [out] uint64& number,
    uint64 i
) cil managed
{
    .maxstack 8
    // number = i;
    IL_0000: ldarg.0
    IL_0001: ldarg.1
    IL_0002: stind.i8
    // return number;
    IL_0003: ldarg.0
    IL_0004: ldind.i8
    IL_0005: ret
}
```

Listing 28: The `OutArg` methods IL code.

The code in Listing 28 is only different from the `Ref` code in regard to names of methods and the presence of `[out]` in the IL for `OutArg`. `[out]` means that the parameter is intended to return a value from the method [35]. This however is not enforced by the Common Language Infrastructure.

Based on this we conclude that `ref` and `out` are the same when modifying values in a method.

Modifying value-type

In Section A.5, Table A.80 we see the package energy of 10 runs of the parameter mechanism benchmarks. The benchmarks have a variance ranging from 10,13% to 14,73%. We see that the package energy of using `return` is a lot higher than using `ref`. To see why this is the case we look at the code for the benchmarks Listing 12 in Section 4.1.3. In the listing we see how the `ref` benchmark functions alongside an explanation of the differences compared to the `return` benchmark. The difference between the benchmarks happens in the for loop. In the `ref` benchmark both parameters for the struct are passed using the `ref` keyword. In the "normal" case the parameters are passed as `obj = ReturnModStruct(obj, i);`, which means they are passed by value. Because of this, it has to be returned from the method and then assigned. The difference here is apparent in how the two benchmarks vary in how they pass the parameters to the methods [34, 32]. The interesting part is the difference between the small and large structs. The pass-by-reference benchmarks have similar results which are expected as passing by reference does not change with the size of the struct. Where in the case of passing by value the size has an effect and it is thus more expensive with a larger struct. To further examine we look at the IL code.

```
1 // BigHelperStruct obj = new BigHelperStruct();
2 IL_0000: ldloca.s     obj
3 IL_0002: call         instance void Benchmarks_P10.benchmarks.
  ↪ ParameterMechanismBenchmarks/BigHelperStruct::.ctor()
4 ...
5 // obj = ReturnModStruct(obj, i);
6 IL_000c: ldloc.0
7 IL_000d: ldloc.1
8 IL_000e: call         valuetype Benchmarks_P10.benchmarks.
  ↪ ParameterMechanismBenchmarks/BigHelperStruct
  ↪ Benchmarks_P10.benchmarks.
  ↪ ParameterMechanismBenchmarks::ReturnModStruct(valuetype
  ↪ Benchmarks_P10.benchmarks.
  ↪ ParameterMechanismBenchmarks/BigHelperStruct, unsigned
  ↪ int64)
```

```
9 IL_0013: stloc.0
```

Listing 29: The `ReturnLargeStruct` IL code.

```
1 // BigHelperStruct obj = new BigHelperStruct();
2 IL_0000: ldloca.s     obj
3 IL_0002: call         instance void Benchmarks_P10.benchmarks.
  ↪ ParameterMechanismBenchmarks/BigHelperStruct::.ctor()
4 ...
5 // RefModStruct(ref obj, ref i);
6 IL_000c: ldloca.s     obj
7 IL_000e: ldloca.s     i
8 IL_0010: call         void Benchmarks_P10.benchmarks.
  ↪ ParameterMechanismBenchmarks::RefModStruct(valuetype
  ↪ Benchmarks_P10.benchmarks.
  ↪ ParameterMechanismBenchmarks/BigHelperStruct&, unsigned
  ↪ int64&)
```

Listing 30: The `RefLargeStruct` IL code.

Looking at the code in Listing 29 and Listing 30 we have three differences. In Listing 29 the instruction `ldloc` is used to load local variables to the evaluation stack [59]. Once the call is done the `stloc` instruction is used to pop this value from the top of the evaluation stack [60]. Listing 30 handles this in a different manner by using the `ldloca.s` instruction to load the address of the two local variables onto the evaluation stack [61]. Another difference is that the `ref` version has an added & at the end of the method call which shows that we are dealing with a reference. Already here there is part of the explanation for lower energy consumption, as fewer instructions are needed. However, as the two benchmarks use different methods we also look at the IL for these methods.

```
1 .method private hidebysig static
2 valuetype
  ↪ Benchmarks_P10.benchmarks.ParameterMechanismBenchmarks/BigHelperStruct
  ↪ ReturnModStruct(
3     valuetype Benchmarks_P10.benchmarks.
      ↪ ParameterMechanismBenchmarks/BigHelperStruct obj,
```

```
    unsigned int64 i
) cil managed
{
    .maxstack 8
    // obj.Number += i;
    IL_0000: ldarga.s     obj
    IL_0002: dup
    IL_0003: call         instance unsigned int64
    ↪  Benchmarks_P10.benchmarks.
    ↪  ParameterMechanismBenchmarks/BigHelperStruct::get_Number()
    IL_0008: ldarg.1
    IL_0009: add
    IL_000a: call         instance void
    ↪  Benchmarks_P10.benchmarks.
    ↪  ParameterMechanismBenchmarks/BigHelperStruct::set_Number(unsigned
    ↪  int64)
    // return obj;
    IL_000f: ldarg.0
    IL_0010: ret
}
```

Listing 31: The`ReturnModStruct` IL code.

```
.method private hidebysig static
void RefModStruct(
    valuetype Benchmarks_P10.benchmarks.
    ↪  ParameterMechanismBenchmarks/BigHelperStruct& obj,
    unsigned int64& i
) cil managed
{
    .maxstack 8
    // obj.Number += i;
    IL_0000: ldarg.0
    IL_0001: dup
    IL_0002: call         instance unsigned int64
    ↪  Benchmarks_P10.benchmarks.
    ↪  ParameterMechanismBenchmarks/BigHelperStruct::get_Number()
    IL_0007: ldarg.1
```

```
13      IL_0008: ldind.i8
14      IL_0009: add
15      IL_000a: call         instance void
        ↪  Benchmarks_P10.benchmarks.
        ↪  ParameterMechanismBenchmarks/BigHelperStruct::set_Number(unsigned
        ↪  int64)
16      // }
17      IL_000f: ret
18  }
```

Listing 32: The`RefModStruct` IL code.

Examining the two methods called Listing 31 and Listing 32 we find multiple differences. The first difference is the first instructions in each method. Listing 31 calls `ldarga.s` where as Listing 32 calls `ldarg.0`. In this case `ldarga.s` loads an argument from an address and puts it on top of the evaluation stack, where as `ldarg.0` loads the argument at index 0 onto the evaluation stack [62]. The `ref` benchmark has the additional instruction `ldind.i8`, which indirectly loads a value of type `int64` onto the evaluation stack [63]. While the `return` benchmark has another `ldarg.0` instruction before returning, which loads the argument to index 0, which is then returned by `ret`[64]. The difference here is that the `return` benchmark returns by value, while the `ref` benchmark returns by reference. This explains the different energy consumption that is observed between the two.

Returning

Given the results seen in Section 4.1.3 show near to no difference between benchmarks, we analyze the benchmarks to why there is no difference.

The benchmarks test use of different language constructs in the form of `ref`, `out`, and `return`. The difference between the benchmarks is the helper methods that are called in each benchmark. There is a difference in the helper methods, which does not explain why the result of running these benchmarks are so similar. To explain the results we examine the `IL` code of the benchmarks.

Examining the `IL` code we find that the `ref` and `out` benchmarks have identical `IL` code.

```
    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: ldarg.1
    IL_0003: ldarg.1
    IL_0004: add
    IL_0005: stind.i8
    IL_0006: ret
}
```

Listing 33: TheReturnRefHelp IL code.

In Listing 33 the IL code for the ReturnRefHelp method called in the ref benchmark is seen, the code for the out benchmark is identical. Looking at the IL code of the helper methods we see differences between the ref, out and the two versions of return.

```
IL_0000: nop
    IL_0001: ldarg.1
    IL_0002: ldarg.1
    IL_0003: add
    IL_0004: starg.s      number
    IL_0006: ldarg.0
    IL_0007: stloc.0
    IL_0008: br.s         IL_000a
    IL_000a: ldloc.0
    IL_000b: ret
}
```

Listing 34: TheReturnHelp IL code.

```
IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: ldarg.0
    IL_0003: add
    IL_0004: stloc.0
    IL_0005: br.s         IL_0007
    IL_0007: ldloc.0
    IL_0008: ret
}
```

Listing 35: TheReturnHelpAlt IL code.

Listing 34 and Listing 35 shows the two helper methods used in the `return` benchmarks. The difference is that in the standard version the addition is stored in a specified argument slot. This adds an additional two instructions to load the argument at index 0 of the evaluation stack, and then add it to the local variable list. These extra instructions are `IL_0006` and `IL_0007` in Listing 34. The alternative version pops the result to the local variable list at index 0 directly at `IL_0004` in Listing 35.

Looking at the `IL` code we see that the `ref` and `out` benchmarks are identical and both are pass by reference. The two benchmarks which use the return construct are passed by value. The identical IL code explains why there should be no difference between the benchmarks. We have from Section 4.1.3 and Section 4.2.3, that with larger objects there is a significant difference in the energy consumption between these different approaches, from this we conclude that in this particular case the size of the objects are not of a substantial size in which a difference between the approaches can be measured.

4.2.4 Lambda

With our `lambda` benchmarks we look into why there is a variance in the energy consumption of the different results. And why closure affects energy consumption. First, we look at what the average package energy consumption has been across the 10 initial runs.

Variance

We see in Table A.83 from Section A.5 the results of the first 10 runs, we can also see the variance between the different benchmarks varies. All of the benchmarks exhibit varying behavior, therefore we hypothesize the variance stems from something the benchmarks have in common. These benchmarks have two common attributes, namely using a variation of a `delegate`, and using lambda expressions. Based on this we first look into `delegate` and determine if this is the cause of the variance.

Per the documentation [65, 66, 48] `Func<>` and `Action<>` utilize `delegate`.We look at the documentation for `delegate` [37] to see if there is an explanation for the variance. Per the documentation "...delegates encapsulate both an object instance and a method." [37], this leads us to the hypothesis that the encapsulation of object instances could cause us to need more frequent garbage collection. In order to test this hypothesis we run the benchmarks once more, however we make use of `GC.TryStartNoGCRegion` [53] and `GC.EndNoGCRegion` [54], to eliminate the effect of garbage collection.

We see in Section A.5, Table A.84 that there is a large variance in the results. We find that the variance has both increased and decreased across the different benchmarks. It has decreased for `Lambda Delegate Closure`, increased for `Lambda Action`, `Lambda Delegate`, `Lambda Closure` and `Lambda Param`. For the most part, turning off garbage collection has resulted in greater variance. Based on this we can not conclude that garbage collection is the cause of the large variance. Therefore we create variations of the benchmarks which use functions instead of lambda expressions, to test the second common attribute.

```
1  public class LambdaLocalFuncBenchmarks {
2      ...
3      [Benchmark("No Lambda Expression", "Tests a lambda
   ↪   expression using Func<>")]
4      public static ulong Lambda() {
5              ulong result = 0;
6              Func<ulong> testFunc = FuncTest;
7              for (ulong i = 0; i < LoopIterations; i++) {
8                      result = result + testFunc() + i;
9              }
10             return result;
11     }
```

```
12      ...
13      public static ulong FuncTest() {
14              return 10;
15      }
16      ...
17  }
```

Listing 36: The `Lambda()` method tests `Func<>` without a lambda expression.

In Listing 36 we see an example of an altered benchmark. In this particular instance the original `Lambda` benchmark now uses a method in line 6 instead of `Func<ulong> testFunc = () => 10;`. These changes have been done for all of the original lambda benchmarks.

Section A.5, Table A.85 shows the result across 10 runs of the new benchmarks and the variance that has been observed between these runs. The percentage variance observed here is comparable to what was seen with the initial benchmarks in Table A.83, except for the `Lambda` benchmark, however, because the other benchmarks still have the same behavior and all utilize lambda, we assume this to be an outlier. Based on this observation we rule out the lambda expressions themselves as the cause of the observed variance.

Thus, the cause is likely because of the interaction between delegates and memory. To test this hypothesis we run one of our benchmarks again, however, this time we run only a single benchmark. We choose to run the `Lambda` benchmark. The reason we only run one benchmark is how caches and branch predictors make the performance dependent on machine-specific parameters and the exact layout of code, stack frames, and heap objects. By only running one benchmark we can reduce how the layout of code, stack frames, and heap objects are impacted. The benchmark is then performed both with garbage collection on and off.

Looking at Section A.5, Table A.86 we can see that the variance is still high, in fact the variance is comparable to what was found in Table A.83. As such this does not seem to be the cause of the observed variance.

We run the benchmark one more time without garbage collection, to see if there are any differences from what we initially observed for turning off garbage collection. When we look at the results in Table A.87 in Section A.5 compared to Table A.86 we see that there is still a large amount of variance in the results, however, it has been significantly reduced. While the results are still not consistent the results imply that the variance is in large

part caused by garbage collection. It might be the case, based on the documentation for `GC.TryStartNoGCRegion` [53] and `GC.EndNoGCRegion` [54], that garbage collection still occurs in some cases and thus getting results that have lower variance by reducing garbagre collection is difficult.

We perform two more experiments to see if we can eliminate the variance. These experiments consist of using ReadyToRun (R2R) compilation instead of Just-In-Time (JIT) for the `Lambda` benchmark, both with and without garbage collection. These experiments are conducted as the large variance might have to do with the JIT compilation of the lambdas, and using R2R compilation reduces the amount of work done by the JIT compiler. The documentation however states that frequently used methods may still be compiled and replaced by a JIT compiled version during runtime [67].

Looking in Section A.5 at the results in Table A.88 compared to Table A.87 and Table A.86, we see the variance is approximately 9%. This is similar to what we found by turning garbage collection off. We also run these with garbage collection turned off, to test if this gives consistent results.

In Section A.5, Table A.89 we see the variance is actually about 0,5% larger than in Table A.88 where garbage collection is not disabled. This implies that the effects of JIT compilation compared to R2R are similar to the difference between JIT with and without garbage collection. From this, we conclude that garbage collection is likely causing most of the variance.

Closure

In Section 4.1.4 we found that using closure is less efficient than not using it, so we now look into why there is a difference between these results. First, we look at the differences in the benchmarks, where `Lambda Closure` is shown in Listing 15, alongside this listing there is an explanation of it and the differences it has with the `Lambda` benchmark. The difference between them is that the `Lambda` benchmark does not use closure and therefore it does not capture the value of `result`, nor does it utilize it in the lambda expression. To have the same results the call of the lambda expression is different, with a call and addition of the variable which is captured in the closure version. As this is the only difference and it is both needed and expected to make use of lambda closure, it is likely the cause of differing energy consumption.

Looking at the documentation for lambda expressions, in the section "Capture of outer variables and variable scope in lambda expressions" [36].

We find that the variables that are captured are stored in the lambda expression even if they would normally be garbage collected. It is not garbage collected until the delegate which references it is garbage collected itself. From this, we have that lambda using closure is not only subject to the garbage collection of the delegates used for the expression but also subject to garbage collection of the captured variable. As we have found garbage collection has a significant impact on the energy consumption of lambda expressions, increasing the amount of garbage collection needed also increases the energy consumption. This explains why lambda expressions that use closure consume more energy than their counterparts.

4.2.5 Summary

In general, some degree of variance should be expected for all language constructs. This variance can be both an inherent part of the language construct itself, but it can also come from external processes.

- `Concurrency`
  - Concurrency can result in a significant speedup and reduction in energy consumption.
  - The implementations of `Parallel.For`, `ThreadPool`, and the `Task` have overhead as they need to be generic.
  - Overhead becomes insignificant as the workload gets larger.
  - `Parallel.For` is the most efficient of the constructs.
  - `ThreadPool` is the most efficient construct for doing many different tasks.
- `Casting`
  - In the options for casting there is no difference in effect for most cases.
  - Behavior is caused by implementations being the same.
  - Explicit and implicit casting have the same implementation.
  - Explicit casting and `as` have the same energy consumption because `isinst` and `Castclass`, have identical behavior [56].
- `Parameter Mechanisms`

  - Parameter mechanisms we have looked at have no difference in effect because of identical implementations.
  - Garbage collection causes some of the variance observed, in the non-modifying group.
  - `ref` is well suited for modification and return of value-types, because values are passed by reference.
  - `in` and `ref` have identical IL code.
  - Modifying reference types with `ref` and `out` is only different in the `[out]` keyword in the IL code.
  - Returning values using `ref` and `out` instead of `return`, `ref` and `out` has no tangible difference in energy consumption for small arguments.

- `Lambda`
  - Garbage collection causes some of the variance observed.
  - Using delegates and closures has extra costs in the form of garbage collection.

Following this analysis and the results we have gathered for the different language constructs, we utilize this information in the creation and presentation of the results of our macrobenchmark. Several of the examined language constructs are identical in energy behavior, therefore we look at the work in [3] as a source of other language constructs that we can use for constructing our macrobenchmarks and their permutations.

Chapter 5

Macrobenchmarks

This chapter presents our work regarding macrobenchmarks. First, a presentation is given of the results of running the macrobenchmarks in their original state and optimized state using the hardware configuration shown in Table 3.1, Section 3.3.

5.1 Selection Process

To determine the effects of different language constructs and combinations thereof we select benchmarks capable of utilizing the language construct changes and combinations we test.

Our macrobenchmarks are collected from *Rosetta Code* [2]. We choose to use *Rosetta Code* as it offers a wide range of benchmarks, solving many different tasks. As we work with C# *Rosetta Code* presents us with 906 different tasks. This amount of tasks and their wide range of focus allows us to cover multiple different areas and thereby ensure that we test benchmarks where we can cover several different language constructs in combination.

We must also determine which language constructs to use. In Section 4.1 and Section 4.2, we found that several of the language constructs we look at have the same energy consumption. This limits what language constructs from this project are applicable. To help with this we create a list of language construct changes based on the results gathered in this project and in [3]. The list consists of eight groups, which are `Concurrency`, `Parameter Mechanisms`, `Lambda`, `Datatypes`, `Selection`, `Loops`, `Collections` and `Objects`. The list is seen in Section A.10. This list is not a strict guide, where every section must be changed in the macrobenchmarks. Every mac-

robenchmark is subject to three changes as described in Section 3.4.

As concurrency is the focus of our explored constructs it is important that the macrobenchmarks we select, are already using concurrency or can be modified to make use of it. However while important, we do allow benchmarks that neither have nor can be converted to make use of concurrency.

The benchmarks must contain three of the categories presented in Section A.10 such that we can change them.

Based on the list of changes and the requirements for our benchmarks we have selected the following 8 macrobenchmarks, which are described in Section 5.2.

- 100 prisoners
- Almost Prime
- Chebyshev coefficients
- Compare length of two strings
- Dijkstra's Algorithm
- Four rings or Four squares puzzle
- Numbrix Puzzle
- Sum to Hundred

These benchmarks are selected based on an analysis that determines if they are suitable for concurrency-related changes. One of the benchmarks does not fit this specification, *Dijkstra's Algorithm*, which is done to explore more changes that do not involve concurrency.

When looking at other possible changes for a benchmark besides concurrency, we use an experimental approach where the different groups from Section A.10 are tested. This testing is done by implementing the changes from that particular category. If the changes are possible without changing the result of the original benchmark, that group is used for the benchmark, and we then have another variation of the original benchmark.

5.2 Results

In this section, we present the macrobenchmarks, changes made to them, and the results of running them, important to note is that all results are normalized to 1.000.000 iterations [3, p.60]. All the benchmarks can be found in our git repository [1]. The process of presentation is as follows:

- Present the benchmark and its functionality, any changes we are making to it if necessary, and what language constructs we are changing.
- Show an example of how code has changed for each of the language constructs that are changed.
- Show a plot of the baseline energy consumption of the macrobenchmark, alongside the energy consumption of all its permutations.
- Present the numerical results for baseline benchmark and permutations, for comparison.
- Perform sanity checks to ensure that the energy consumed by both the baseline benchmark and its permutations is plausible.
- Summarize our findings, about which combination of changes is better.

5.2.1 100 Prisoners

The benchmark [68] is an implementation of the 100 prisoners' problem. This problem revolves around 100 prisoners finding their own number in 100 drawers, where they may only open 50 drawers and are not allowed to communicate with other prisoners. If all 100 prisoners find their number, they are pardoned. In the `100 Prisoners` benchmark we change `Concurrency`, `Collections` and `Loops` from Section A.10.

We also change parts of the benchmark, to allow it to fit the structure of benchmarks in the framework [69]. We change the number of executions of the problem itself in the benchmark, we change what the output of running the benchmark is, and we change the technique used for sorting an array randomly. The number of executions inside the benchmark is changed from 1.000.000 to 100.000, we make this change because we have found that running the benchmark takes approximately four minutes per iteration. Given that `100 Prisoners` is expected to run a minimum of 10 iterations, to determine how many iterations are needed for significant results and to normalize return values, `100 Prisoners` exceeds the limit of 45 minutes per benchmark defined in the `Benchmark` class of the framework [69]. The change in output is done as the original benchmark prints the results, we instead return these results, which allows us to compare the results of the different permutations of the original benchmark. The random technique is changed to make the "randomness" consistent between

runs, as the original benchmark uses `GUID.NewGuid()`, and we change it to use the `Random` class, as this can be instantiated with a seed that makes it consistent. These changes can be seen in our GitLab repository [1].

Concurrency

For the `Concurrency` group of language constructs, we change how multiple executions of the problem are run in parallel, whereas the original version of the benchmark runs each execution sequentially.

```
1  public class PrisonersParallelFor {
2      ...
3      static double Exec(uint n, Func<int, bool> play) {
4              uint success = 0;
5              Parallel.For(0, (int)n, index => {
6                      if (play(index)) {
7                              Interlocked.Increment(ref success);
8                      }
9              });
10             return 100.0 * success / n;
11     }
12     ...
13 }
```

Listing 37: Change to Default implementation of `100m Prisoners`, Exec method.

In Listing 37 we can see how the Exec method looks after it has been converted to a concurrent version. Instead of using a regular `for` loop we now use `Parallel.For` and the number of successful executions is incremented using `Interlocked.Increment` instead of `success++`. Also, the random seed, being the index of the loop, has to be passed as a parameter to the functions instead of being a class variable, as different threads would otherwise interfere with each other's seeds.

Loops

Using Language Integrated Query (LINQ) has proven to be slow and energy consuming [3]. We change the benchmark to avoid using LINQ, in accordance to the `Loops` section of Section A.10.

```
1  public class PrisonersNoLinq {
2      ...
3      static bool PlayOptimal() {
4              var rnd = new Random(_seed);
5              var keys = new int[100];
6              for (int i = 0; i < 100; i++) {
7                      keys[i] = rnd.Next();
8              }
9              var secretsArr = new int[100];
10             for (int i = 0; i < 100; i++) {
11                     secretsArr[i] = i;
12             }
13             Array.Sort(keys, secretsArr);
14             var secrets = new List<int>(secretsArr);
15         ...
16     }
17     ...
18 }
```

Listing 38: Change to Default implementation of `100 Prisoners`, PlayOptimal method.

In Listing 38 we see the changes made to the `PlayOptimal` method after removing usage of LINQ. What was originally `Enumerable.Range(0, 100).OrderBy(a => rnd.Next()).ToList();` has been changed to what we see in line 5 to 16. First, an `array` containing random numbers is made, which is used as the ordering when sorting. Then, an `array` containing the numbers from 0 to and including 99 is created. The latter array is then sorted based on the ordering of the random numbers in the first array. Finally, a `List` is instantiated with these random values, as the original benchmark uses a `List`. This conversion happens as we are not looking at changing the type of collection used in this version of the benchmark, and `List` does not have the specific sort function used. The benchmark contains another method called `PlayRandom`, which is changed in the same manner.

Collections

We change the benchmark to use an `array` instead of a `list`. The changes to the benchmark depend on whether the benchmark uses LINQ or not. If the benchmark does use LINQ the `ToList()` call is simply changed to `ToArray()`. If the benchmark uses LINQ the redundant call to `var secrets = new List<int>(secretsArr);` in Listing 38 is removed, and `secretsArr` is renamed to `secrets`. These changes are based on what is seen in Listing 38 as part of the work of changing collections is already accomplished.

```
1 public class PrisonersNoLinq {
2     ...
3     static bool PlayOptimal() {
4             var rnd = new Random(_seed);
5             var secrets = Enumerable.Range(0, 100).OrderBy(a =>
              ↪  rnd.Next()).ToArray();
6         ...
7     }
8     ...
9 }
```

Listing 39: Change to Default implementation of `100 Prisoners`, `PlayOptimal` method.

Listing 39 shows us the how the `PlayOptimal` method looks after the `lists` have been exchanged for `arrays`. In line 5 the `secrets` variable is now an array, as `ToArray()` is used instead of `ToList()`. This change is also done for the `PlayRandom` method.

As mentioned before these changes are then combined to allow us to see which combination of changes has the lowest energy consumption. This means that we have eight versions of the `100 Prisoners` benchmark. These are the three seen here and the original from [68], alongside their combinations.

Results

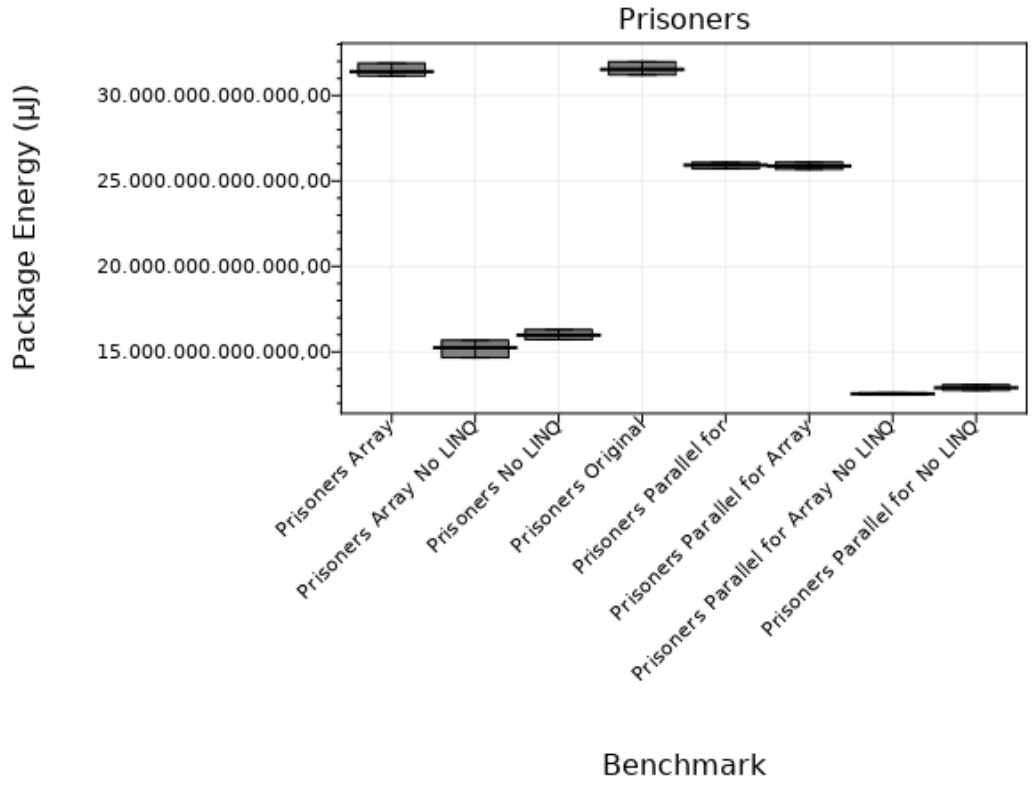

Figure 5.1: The energy consumption of 100 Prisoners. The y-axis starts at 12.000.000.000.000 μJ and ends at 33.000.000.000.000 μJ.

In Figure 5.1 we see the energy consumption of the eight different variations of `100 Prisoners`. we observe that the elimination of LINQ is present in all of the benchmarks with heavily reduced energy consumption. We also see the use of `Parallel.For` has decreased the energy consumption by a substantial amount, which fits with our results from Section 4.1.1. This is consistent across the 10 runs of the benchmarks. To see the variance observed for the benchmarks see Section A.6, Table A.90. We subject the results to Kolmogorov-Smirnov tests, which are seen in Section A.9, Table A.121.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Package Energy Change from Original
Prisoners Array	2.537.415.484,62	31.462.627.000.000,00	1.416.101.230.769,23	- 0,24 %
Prisoners Array No LINQ	1.323.478.131,58	15.275.737.026.315,79	738.448.000.000,00	- 51,56%
Prisoners No LINQ	1.348.765.992,86	16.007.505.571.428,57	753.699.428.571,43	- 49,24%
Prisoners Original	2.573.008.333,33	31.538.132.250.000,00	1.436.545.250.000,00	Not Applicable
Prisoners Parallel for	695.749.820,00	25.907.526.500.000,00	397.209.700.000,00	- 17,85%
Prisoners Parallel for Array	702.119.200,00	25.876.201.384.615,38	402.408.461.538,46	- 17,95%
Prisoners Parallel for Array No LINQ	363.827.100,00	12.538.147.692.307,69	208.157.461.538,46	- 60,24%
Prisoners Parallel for No LINQ	378.939.509,52	12.896.718.761.904,76	218.493.619.047,62	- 59,11%

Table 5.1: Table showing the elapsed time and energy measurement for each Prisoners.

In Table 5.1 we see the numerical values of the results. Their accompanying p-values are seen in Section A.3.1. Examining Table 5.1 we see that all changes lead to a decreased energy consumption. Additionally, the size of this effect ranges substantially from the original benchmark, with `Prisoners Array` only having an effect of -0,24%, compared to the change of -60,24% in `Prisoners Parallel for Array No LINQ`. We see that `Prisoners Parallel for` changes energy consumption by -17,85%, which combined with the substantial reduction of elapsed time, shows the dual benefits of utilizing concurrency. This shows that removing LINQ has the largest impact on energy consumption and combining it with other changes that reduce energy increases this effect. We see that in combination with the results of the Kolmogorov-Smirnov tests that all results are part of different distributions, except for `Prisoners Parallel for` and `Prisoners Parallel for Array` where changing to an array is the only difference. Adding the information presented in Table A.90, we see that on average the energy consumption is increased by 0,60%, compared to `Prisoners Parallel for` without the array. This change is however insignificant compared to the overall energy consumption, and the two are determined to be part of the same distribution. We can conclude that all changes have an effect that reduces energy consumption compared to the original benchmark.

We check the plausibility of these results by performing sanity checks. We first check if the measurement from the original benchmark is plausible.

$$2.573.008.333,33 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 40.730.721.916.666,60\mu\text{J} \quad (5.1)$$

Equation 5.1 shows our estimated energy consumption for the original version of `100 Prisoners`. This value and the one by the benchmark have a difference of 25,44%, which is a plausible difference.

We perform a second sanity check, on the benchmark with the lowest energy consumption.

$$363.827.100,00 \text{ ms} \cdot 1000 \cdot 4 \text{ cores} \cdot 15,83 \text{ Watts} = 23.037.531.972.000,00\mu\text{J} \tag{5.2}$$

Equation 5.2 shows the estimate of `Prisoners Parallel for Array No LINQ`. Compared to the recorded result, we have a difference of 59,03%, which is a plausible difference.

Findings

The most crucial finding here is how all the different versions are better if `LINQ` is removed. Beyond that, all other changes reduce energy consumption and continue to do so, when they are combined with other changes, except for changing the collections to arrays, which has a negligible effect when only combined with parallelization.

- Concurrency has a significant effect on energy consumption and reduces elapsed time considerably.
- Loop changes in the form of LINQ lead to a significant reduction in the energy consumption of the benchmark.
- Collection changes from a `List` to an array do not significantly reduce energy consumption when used in conjunction with parallelization, which is surprising considering the findings in[3].
- Effect of all changes is increased when added together, except when using `Parallel.For` and arrays only, which is surprising as it is the only combination that exhibits this behavior.

5.2.2 Almost Prime

The benchmark [70] is an implementation of a method for generating numbers from k prime numbers, these numbers are called Almost primes. A table is then created for the first ten k-Almost primes. This is done 5 times, where k goes from 1 to 5. We look at changing `Concurrency`, `Collections` and `Objects` from Section A.10.

The following changes are made to the original benchmark. Instead of printing the results to the console, we sum the results and return the value. We also increase the limit for `k` to 10, to increase the workload. These changes are seen in our GitLab repository [1].

Concurrency

We change part of the benchmark to make use of manual threading. This is done in the part of the benchmark which is responsible for finding the first 10 results per `k`.

```
public class AlmostPrimeManualThread {
    ...
    public List<int> GetFirstN(int n) {
            ConcurrentBag<int> result = new
            ↪  ConcurrentBag<int>();
            var pCount = Environment.ProcessorCount;
            var threads = new Thread[pCount];
            int num = 1;
            for (int i = 0; i < pCount; i++) {
                    threads[i] = new Thread(_ => {
                            while (result.Count < n) {
                                    var val =
                                    ↪  Interlocked.Increment(ref
                                    ↪  num);
                                    if (IsKPrime(val)) {
                                            result.Add(val);
                                    }
                            }
                    });
                    threads[i].Start();
            }
            foreach (var thread in threads) {
                    thread.Join();
            }
            var res = new List<int>(result);
            res.Sort();
            var diff = res.Count - n;
            if (diff > 0) {
                    res.RemoveRange(n, diff);
            }
            return res;
    }
    ...
```

```
31 }
```

Listing 40: Change to Default implementation of `Almost Prime`, GetFirstN method.

In Listing 40 we see the changes after the benchmark is converted to a concurrent version. Instead of using a regular `for` loop we use an array of threads and the task that is to be completed is added to these. Once completed we clean up the result to have the same order and ensure that entries in the array which are not used are set to zero.

Collections

The benchmark is changed to use an array instead of a `List`. This is done in the same manner as for `100 Prisoners`. We have two changes to the benchmark, this being the `Run` method, which uses a `foreach` loop to iterate through results instead of calling `ForEach` on a `List`. The `GetFirstN` method is also changed to return an array instead of a `List`, and the loop within the method uses an extra variable `primesCount` to keep track of the number of primes found and the index of the array which primes are added to. The changes are seen in the GitLab repository [1].

Objects

The last change made to the benchmark comes in the form of changes to the objects. We change the class `KPrime` to be a struct. The changes can be seen in our GitLab repository [1].

All these changes are combined to see how the different changes affect one another and what combinations have the highest and lowest energy consumption.

Results

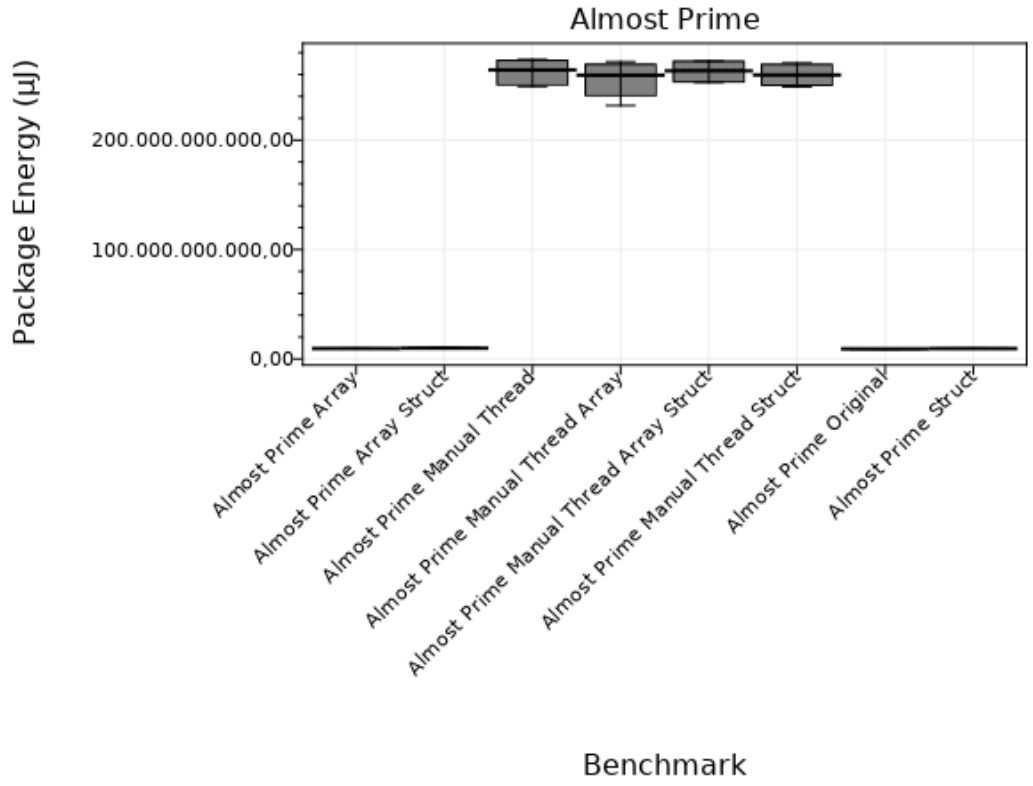

Figure 5.2: The energy consumption of Almost Prime. The y-axis starts at 0 μJ and ends at 280.000.000.000 μJ.

In Figure 5.2 we can see the energy consumption of the eight different variations of `Almost Prime`. We see that for all benchmarks which utilize concurrency the energy consumption is higher than the original version. This is different from what was found in Section 4.1.1, where manual thread was found to be a good choice no matter the amount of work that is performed. We look into why there is this difference compared to what was initially found when we move on to the analysis of the results. Another thing to note here is how all the changes that do not involve concurrency look to be similar to the energy consumption of the original benchmark. The variance of these results are seen in Section A.6, Table A.91. The results of the Kolmogorov-Smirnov are found in Table A.122, Section A.9.2.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Change from Original
Almost Prime Array	863.834,22	9.482.240.234,38	480.900.976,56	+ 3,29%
Almost Prime Array Struct	892.264,36	9.846.781.445,31	496.612.695,31	+ 7,26%
Almost Prime Manual Thread	15.996.794,04	263.587.480.416,67	9.276.785.833,33	+ 2771,31%
Almost Prime Manual Thread Array	15.983.079,97	258.866.513.849,43	9.268.455.255,68	+ 2719,89%
Almost Prime Manual Thread Array Struct	15.970.885,48	263.821.768.995,10	9.260.377.450,98	+ 2773,89%
Almost Prime Manual Thread Struct	16.055.401,24	259.637.167.564,66	9.316.025.862,07	+ 2728,28%
Almost Prime Original	870.677,38	9.180.029.557,29	484.910.416,67	Not Applicable
Almost Prime Struct	862.159,39	9.457.325.390,62	480.555.273,44	+ 3,02%

Table 5.2: Table showing the elapsed time and energy measurement for each Almost Prime.

In Table A.51 we see the numerical values of the results. The p-values are seen in Section A.3.2. We see that none of the changes have led to decreased energy consumption. Looking at the effect of the different changes we have two distinct regions of increased energy consumption. We have all the variations that make use of manual threading, which range around + 2750%, and then we have `Array` and `Struct` which increase the energy consumption by about 3% each, an effect that stacks once used together.

Examining the results in conjunction with the Kolmogorov-Smirnov tests we find that the original version and the ones that use arrays and `structs` are potentially from the same probability distribution. This fact combined with the small percentage change and the similar energy consumption seen across all ten runs, in Table A.91, we can conclude that in this particular case that these do not affect energy consumption, which is also the case when they are combined with manual threading. For changes with manual threading, the Kolmogorov-Smirnov tests tell us that they are likely part of a different distribution than the original benchmark, but all combinations with manual threading are possibly part of the same distribution, which leads us to the conclusion that they have a significantly large negative impact on energy consumption.

We now look at the plausibility of these results using our sanity checks. First, we have if the measurement from the original benchmark is plausible.

$$870.677,38 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 13.782.822.969,91 \mu\text{J} \tag{5.3}$$

In Equation 5.3, we see the estimated energy consumption for the original benchmark, and we find a difference of 40,09% between it and what is measured, which is a plausible difference.

Next, we perform a sanity check on the benchmark with the highest energy consumption.

$$15.970.885,48 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} \cdot 4\text{Cores} = 1.011.276.468.463,22 \mu\text{J} \tag{5.4}$$

For the estimated value of `Almost Prime Manual Thread Array Struct` and the recorded result, we have a difference of 117,24%. We deem this as plausible based on the calculation being an estimate, we are aware that the extra cores do not all perform the same amount of work, and not all cores are active for the entirety of the benchmark.

Findings

We have from these results that not all changes which should improve energy efficiency according to microbenchmarks improve energy consumption. The most crucial finding in this section is how the use of concurrency can substantially increase energy consumption. While the reason for this is unknown, it will stand as part of the analysis of these results, presented in Section 5.3.

- Concurrency can lead to heavily increased energy consumption alongside increased execution time, which is a surprising result considering the findings in Section 4.1 and what was just found in Section 5.2.1.
- Collection changes in the form of exchanging a `List` for an array do not have a significant impact on energy consumption, which is contrary to the initial findings from [3].
- Object changes in the form of exchanging a `class` for a `struct` does not reduce energy consumption, which is unexpected given previous results from [3].

5.2.3 Chebyshev Coefficients

The Chebyshev coefficients benchmark [71] is an implementation that calculates the Chebyshev coefficients. The exact purpose of the benchmark is not clear according to *Rosetta Code*, as the term Chebyshev coefficients are associated with multiple topics in approximation theory. This is however not relevant to our test, and as such we still use it. We look at changes to `Concurrency`, `Collections` and `Datatypes` from Section A.10.

We change the original benchmark by increasing the value of `nX` from 20 to 100.000.000, which increases the number of approximations performed in the benchmark. We increase this number to have a benchmark with a large workload. This comes from the fact that we have observed different behavior regarding concurrency based on the workload. We also change

the output from being written to console, to instead returning the value of `x`. The full changes are seen in our GitLab repository [1].

Concurrency

In regard to concurrency we change a `for` loop and instead use manual threading, in the same manner as shown in Section 4.1.1 and the `Almost Prime` benchmark in Section 5.2.2. The original version only saves the last result of the `for` loop, while the version we have constructed saves the result of each iteration in an array. Both versions return the value of the last iteration. These code changes can be seen in the GitLab repository [1].

Collections

For changes to Collections, we replace the use of `List` with the use of an array. Originally the values of the coefficients were stored in a `List` which we change to an array. The `ChebCoef` method is changed to both receive and return an array instead of a `List`. We also remove a `for` loop used to initialize the values of the original `List` to 0.0, as this is handled in the creation of the array. Moving through the elements of the original `List` is also changed from using an enumerator to using a `for` loop and the index of the array to iterate through the elements of the array. Lastly the `ChebEval` method now takes an array instead of a `List`. As with other changes, these are seen in the GitLab repository [1].

Datatypes

The last change we make to the benchmark is changing the datatypes. We change all uses of `int` to `uint`, except ones that would require casting to continue their functionality.

As with all other benchmarks, we combine the changes to get all permutations of our benchmark.

Results

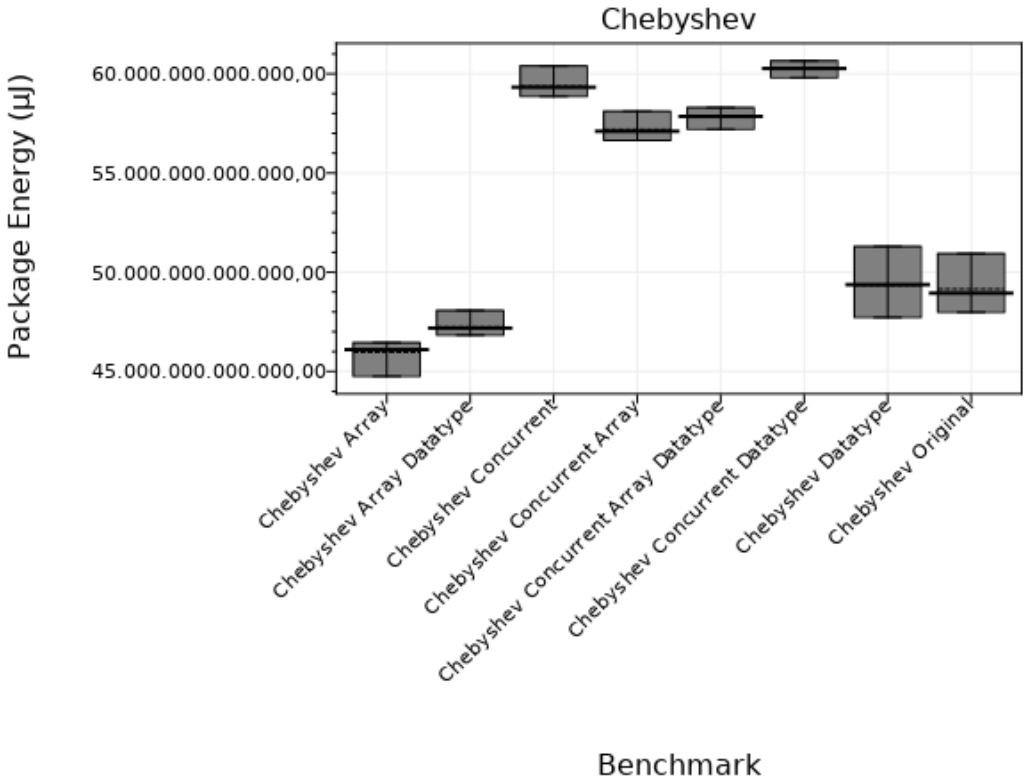

Figure 5.3: The energy consumption of Chebyshev. The y-axis starts at 44.000.000.000.000 μJ and ends at 61.000.000.000.000 μJ.

A noteworthy observation from Figure 5.3 is how all the variations of the benchmark, which make use of `concurrency`, consume more energy than all other versions, similar to what is seen with Almost Prime in Section 5.2.2. The usage of an array instead of a `List` decreases energy consumption while the usage of `uint` instead of `int` yields negligible differences. These effects are consistent across the different combinations, except for the combination with array and datatype changes where datatype changes appear to hurt the results. To see the variance for the benchmarks see Section A.6, Table A.92 and Section A.9, Table A.123 for the Kolmogorov-Smirnov tests.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Change from Original
Chebyshev Array	3.847.904.652,63	45.950.765.105.263,16	2.190.311.631.578,95	- 6,56%
Chebyshev Array Datatype	3.913.061.245,45	47.209.912.272.727,27	2.211.375.818.181,82	- 4,00%
Chebyshev Concurrent	1.678.685.230,00	59.392.086.333.333,336	1.685.109.166.666,67	+ 20,77%
Chebyshev Concurrent Array	1.682.984.922,73	57.172.311.272.727,27	1.667.617.727.272,73	+ 16,26%
Chebyshev Concurrent Array Datatype	1.664.731.190,91	57.870.285.272.727,27	1.674.833.727.272,73	+ 17,67%
Chebyshev Concurrent Datatype	1.661.140.184,62	60.218.567.615.384,62	1.658.710.692.307,69	+ 22,45%
Chebyshev Datatype	3.927.400.347,22	49.352.868.805.555,56	2.226.931.472.222,22	+ 0,36%
Chebyshev Original	3.920.628.134,21	49.178.185.105.263,16	2.220.732.815.789,47	Not Applicable

Table 5.3: Table showing the elapsed time and energy measurement for each Chebyshev.

Table 5.3 shows the numerical values for the different variations of the benchmark. For a look at the p-values of these results see Section A.3.3. Looking at the numbers we see the same pattern as on the plot. We see that the addition of `concurrency` increases energy consumption by 20,77%, and using an array decreases the energy consumption by 6,56%. If we look at the Kolmogorov-Smirnov tests in Section A.9.3 we also see that:

- Original & Datatype
- Concurrent & Concurrent Datatype
- Concurrent Array & Concurrent Array Datatype

are likely to be from the same distributions, while Array & Array Datatype are likely from two separate distributions, along with the rest of the results.

We now check the plausibility of these results by performing sanity checks. We first check if the measurement from the original benchmark is plausible.

$$3.920.628.134,21 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 62.063.543.364.552,50\mu\text{J} \quad (5.5)$$

Equation 5.5 shows our estimated energy consumption for the original version of `Chebyshev`. This value and what is recorded by the framework have a difference of 23,17%, which is a plausible difference.

We perform a second sanity check, on the benchmark with the lowest energy consumption.

$$3.847.904.652,63 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 60.912.330.651.157,90\mu\text{J} \quad (5.6)$$

For the estimate of `Chebyshev Array` and the recorded value, we have a difference of 28,00%, which is a plausible difference.

Findings

The benchmark consumes less energy when using an array instead of `List`, and the impacts of using `uint` instead of `int` are negligible, the effects of using both array and `uint` are worse than just using an array.

- Concurrency significantly increases energy consumption but also cuts runtime in half, the energy consumption increase is surprising as this is different from what the results regarding manual threading in Section 4.1.1 show.
- The usage of an array decreases both energy consumption and runtime.
- The refactoring of datatypes yields negligible changes, which is not unexpected as the changes in energy are relatively minimal compared to the overall energy consumption of the benchmark.
- Combining array and datatype changes reduces the positive effects of using an array, a surprising result considering that array has shown a positive effect and datatype essentially does not affect energy consumption.

5.2.4 Compare length of two strings

This benchmark [72] finds and prints strings in descending order per their length. For this benchmark we change `Concurrency`, `Selection`, and `Loops` from Section A.10.

We make the following changes to the original benchmark. Instead of printing the strings in descending order, they are now returned in the form of a string through the use of `StringBuilder`. The length of this string is then returned as the result of running the benchmark. These changes can be viewed in the GitLab repository [1].

Concurrency

With regard to concurrency, we make two changes, resulting in two different versions of the benchmark. We make two versions, as during the construction of the benchmark variations we found that enacting both of these changes at the same time resulted in a different result than the original version of the benchmark because of a race condition.

Both the changes involve changing `for` loops into `Parallel.For`. The first version makes use of `Parallel.For` to fill an array with the length of strings and their indices, while the second `Parallel.For` version changes the process of determining if a string is the longest, shortest or in between to use a parallel execution instead of a regular `for` loop. These changes are both performed in the `CompareAndReportStringsLength` method.

Selection

In regard to selection we exchange an `if/else` chain with a `switch` statement.

Loops

The last change we make to the benchmark is in regards to `Loops`. In this part we exchange several of the `for` loops in the original benchmark with `foreach` loops. The loops that are changed are the same as the ones we changed regarding concurrency.

Because we have made two different versions of concurrency changes, we end up with more than the regular eight versions of the benchmark. This does not change the process other than giving more variations. We follow the regular process of combining the changes to see how the different changes affect one another and what combinations have the highest and lowest energy consumption.

Results

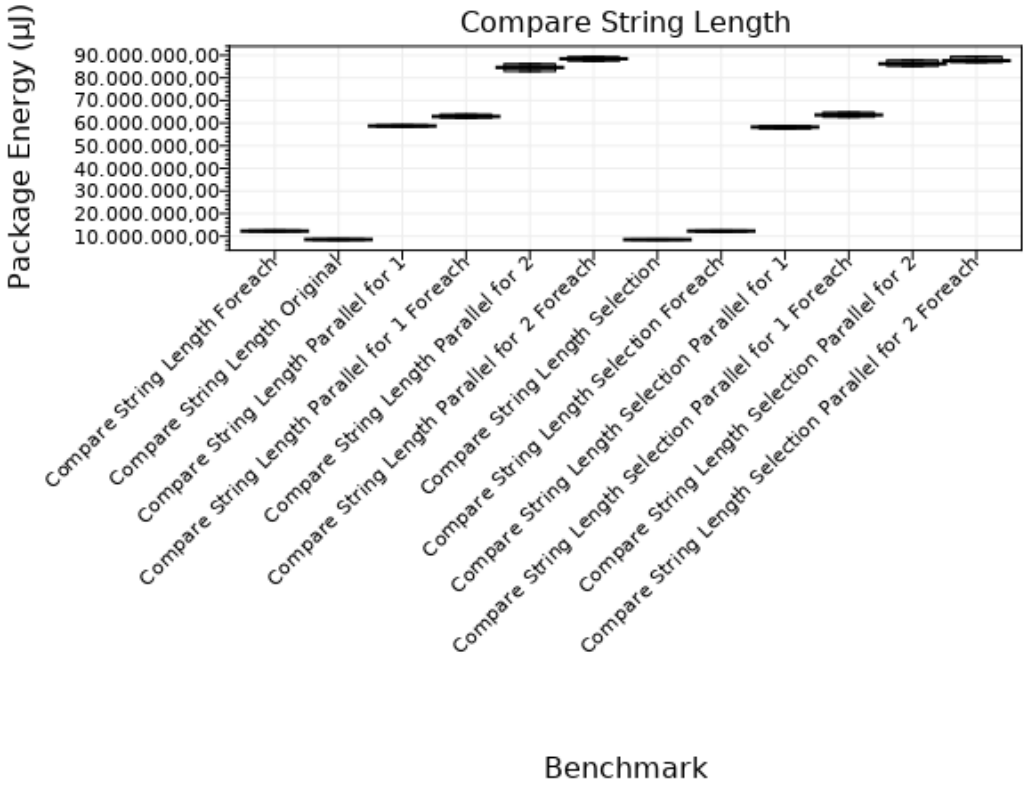

Figure 5.4: The energy consumption of CompareLength. The y-axis starts at 4.000.000 μJ and ends at 92.000.000 μJ.

We see in Figure 5.4 that both usages of `Parallel.For` in this benchmark significantly increases the energy usage. We also see how selection changes are the only changes that seemingly improve energy consumption. The variance of these results are seen in Section A.6, Table A.93. We use the results to perform Kolmogorov-Smirnov tests, the results of these are found in Table A.124, Section A.9.4.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Change from Original
Compare String Length Foreach	889,93	12.308.974,37	513.416,18	+ 44,50%
Compare String Length Original	616,78	8.518.237,45	356.380,16	Not Applicable
Compare String Length Parallel for 1	2.042,82	58.759.381,10	1.175.791,42	+ 589,81%
Compare String Length Parallel for 1 Foreach	2.166,32	63.085.587,72	1.241.056,00	+ 640,59%
Compare String Length Parallel for 2	2.699,55	84.681.729,08	1.573.916,44	+ 894,12%
Compare String Length Parallel for 2 Foreach	2.801,18	88.356.208,80	1.627.622,22	+ 937,26%
Compare String Length Selection	611,84	8.480.392,08	352.702,522	- 0,44%
Compare String Length Selection Foreach	884,24	12.281.944,77	506.595,01	+ 44,18%
Compare String Length Selection Parallel for 1	2.022,75	58.143.856,05	1.172.951,51	+ 582,58%
Compare String Length Selection Parallel for 1 Foreach	2.179,46	63.555.351,80	1.259.244,65	+ 646,11%
Compare String Length Selection Parallel for 2	2.756,75	86.340.315,80	1.610.430,2"	+ 913,59%
Compare String Length Selection Parallel for 2 Foreach	2.790,94	87.879.838,94	1.623.349,67	+ 931,67%

Table 5.4: Table showing the elapsed time and energy measurement for each Compare String Length.

In Table 5.4 we see the numerical values of the results. The p-values are seen in Section A.3.4.

The `Parallel.For` version 1 increases energy by 589,81% while version 2 increases by 894,12%. Converting the two `for` loops to `foreach` loops increases energy consumption by 44,5%, and changing to `switch` statements seems to have negligible results.

According to the Kolmogorov-Smirnov tests adding selection to another variation is likely to be from the same distribution, and thus is unlikely to have a significant effect.

We now check the plausibility of these results by performing sanity checks. We first check if the measurement from the original benchmark is plausible.

$$616,78 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 9.763.690,20\mu\text{J} \tag{5.7}$$

Equation 5.7 shows our estimated energy consumption for the original version of `Compare String Length`. This value and the one by the benchmark have a difference of 13,63%, which is a plausible difference.

We perform a second sanity check, on the benchmark with the lowest energy consumption.

$$611,84 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 9.685.375,88\mu\text{J} \tag{5.8}$$

For the estimate of `Compare String Length Selection` and the recorded result, we have a difference of 12,82%, which is a plausible difference.

Findings

Using `Concurrency` not only increases energy consumption but also run time. Using a `foreach` loop instead of a `for` loop surprisingly also increased energy consumption as opposed to what our testing shows [3].

- Introducing `concurrency` results in significantly higher energy usage as well as run time, which is not unexpected as `Parallel.For` has shown to be inefficient for small workloads.
- Changing a `for` loop to a `foreach` loop increased energy consumption, which is contrary to other results and the findings from [3].
- Swapping `if` statements for `switch` statements has negligible effect, contrary to the assumption that it is the best method for selection.

5.2.5 Dijkstra's Algorithm

The benchmark [73] is an implementation of Dijkstra's algorithm. The benchmark creates a graph with vertices and edges, calculating the shortest path from the starting vertex to the rest.

In the `Dijkstra` benchmark we make changes to `Loops`, `Collections` and `Objects`. We change the output of the original benchmark to return the length of the created string instead of printing it to the console. These changes are seen in our GitLab repository [1].

Loops

In the original benchmark LINQ is used in the `Graph` constructor as well as the `HasEdge`, and `FindPath` methods. We replace the LINQ parts with `for` loops and `foreach` loops where possible.

```
1 public class DijkstraNoLinq {
2 ...
3 public static int Run() {
4 ...
5     var path = graph.FindPath(id('a'));
6     for (int d = id('b'); d <= id('f'); d++) {
7             var localPath = Path(id('a'), d);
8             var stack = new Stack<string>();
```

```
9               foreach ((double distance, int node) in localPath) {
10                  stack.Push($"{name(node)}({distance})");
11              }
12              while (stack.Count > 1) {
13                  sb.Append(stack.Pop());
14                  sb.Append(" -> ");
15              }
16              sb.Append(stack.Pop());
17              sb.AppendLine();
18          }
19      ...
20      }
21  }
```

Listing 41: Change to Default implementation of `Dijkstra's algorithm`, Run method.

We see in Listing 41 an example of the code after we have removed the usage of LINQ. As seen with other benchmark LINQ is replaced through the usage of `for` and `foreach` loops. This particular change is done for the `Run` method.

Collections

The benchmark is changed to use an array instead of a `List`. This is done in the same manner as for `100 Prisoners` and `Almost Prime`. In this regard, we change the `adjacency List` to an array.

Objects

We change the `Graph` class within the benchmark into a `struct` following the guidelines in Section A.10.

Results

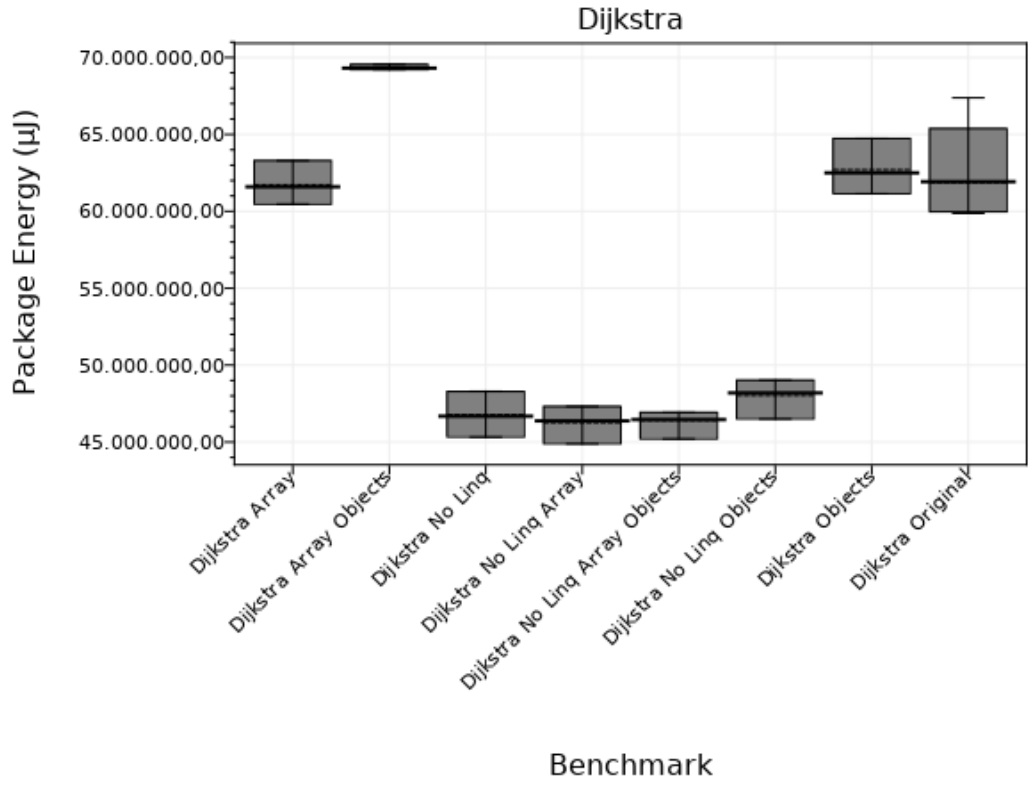

Figure 5.5: The energy consumption of Dijkstra. The y-axis starts at 43.000.000 μJ and ends at 71.000.000 μJ.

We see in Figure 5.5 that every case of replacing the `LINQ` components of the benchmark reduces the energy consumption, as is expected. We can also see how the object change has resulted in increased energy consumption, which is only made worse when combined with the array change. To see the variance observed for the benchmarks see Section A.6, Table A.94. The Kolmogorov-Smirnov tests are seen in Section A.9, Table A.125.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Change from Original
Dijkstra Array	4.396,60	61.662.761,26	2.520.979,98	- 0,29%
Dijkstra Array Objects	5.252,16	69.349.826,05	2.996.429,44	+ 12,13%
Dijkstra No Linq	3.394,66	46.746.571,60	1.928.165,36	- 24,41%
Dijkstra No Linq Array	3.319,59	46.246.371,61	1.886.809,32	- 25,22%
Dijkstra No Linq Array Objects	3.270,73	46.333.644,64	1.862.284,04	- 25.08%
Dijkstra No Linq Objects	3.511,38	48.017.565,26	1.994.329,40	- 22,36%
Dijkstra Objects	4.477,79	62.695.472,13	2.561.846,81	+ 1,38 %
Dijkstra Original	4.522,51	61.842.655,83	2.588.038,99	Not Applicable

Table 5.5: Table showing the elapsed time and energy measurement for each Dijkstra.

In Table 5.5 we see the numerical values of the results. The corresponding p-values are seen in Section A.3.5. The results display savings of energy across the benchmarks. This is especially apparent regarding the benchmarks with changes to `LINQ`. The benchmark where only `LINQ` is changed has a decrease in energy consumption of 24,41%. Surprisingly we see an increase of 1,38% in energy consumption when performing the `Objects` benchmark, and a larger increase of 12,13% when combining the `Array` and `Objects` changes. This increase of 12,13% is quite surprising but when we compare the result in Figure 5.5 with the results from the variance table in Table A.94, we see that the measured result in Figure 5.5 is an outlier. Looking further we see that the Kolmogorov-Smirnov test concludes that the `Array` and `Objects` changes are likely not part of a different distribution than the original benchmark. Surprisingly the combination of the two is significantly different from the original and comparing the resulting mean of the original and the `ArrayObjects` benchmark we calculate an energy increase of 2,69%.

We now check the plausibility of these results by performing our sanity check. We first check if the measurement from the original benchmark is plausible.

$$4.522,51 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 71.591.285,75 \mu\text{J} \tag{5.9}$$

Equation 5.9 shows our estimated energy consumption for the original version of `Dijkstra`. This value and the one by the benchmark have a difference of 15,76%, which is a plausible difference. We perform a similar check with the `No LINQ Array` benchmark which consumes the lowest amount of energy.

$$3.319,59 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 52.549.062,89 \mu\text{J} \tag{5.10}$$

For the estimate of `Dijkstra No LINQ Array` shown in Equation 5.10

and the recorded result, we have a difference of 13,63%, which is a plausible difference.

Findings

The results show that the removal of `LINQ` components decreases the energy consumption. Additionally, the `Objects` version which changes a class to a `struct` causes an increase in energy consumption in the benchmarks where it is applied.

- The use of `LINQ` has a substantial negative effect on energy efficiency.
- The combination of changing `Objects` and `Collections` cause an increase in energy consumption, which given how array and objects do not have a significant impact on energy consumption on their own, is a surprising result.

5.2.6 Four Squares Puzzle

The benchmark [74] is an implementation of the Four Squares Puzzle. The Four Squares puzzle is about placing the letters a through g, each assigned a decimal value, such that the sum of the letters inside each of the squares adds up to the same sum. In the `Four Squares Puzzle` we look at `Concurrency`, `Loops`, and `Selection` from Section A.10.

Instead of printing the different solutions to the console, the total number of solutions is instead summed and returned. These changes are seen in our GitLab repository [1].

Concurrency

In `Four Squares Puzzle` a `for` loop in method `fourSquare` is replaced by a `Parallel.For` loop. This also means that the increment of the number of solutions happens with `Interlocked.Increment()`, to atomically complete the operation, instead of incrementing the variable.

Loops

In `Four Squares Puzzle` a function called `notValid` checks if an integer is present in an array. This is originally done using LINQ but is substituted for a `foreach` loop.

Selection

In `Four Squares Puzzle` three `if` statements in the `fourSquare` method are exchanged for equivalent `switch` statements.

Results

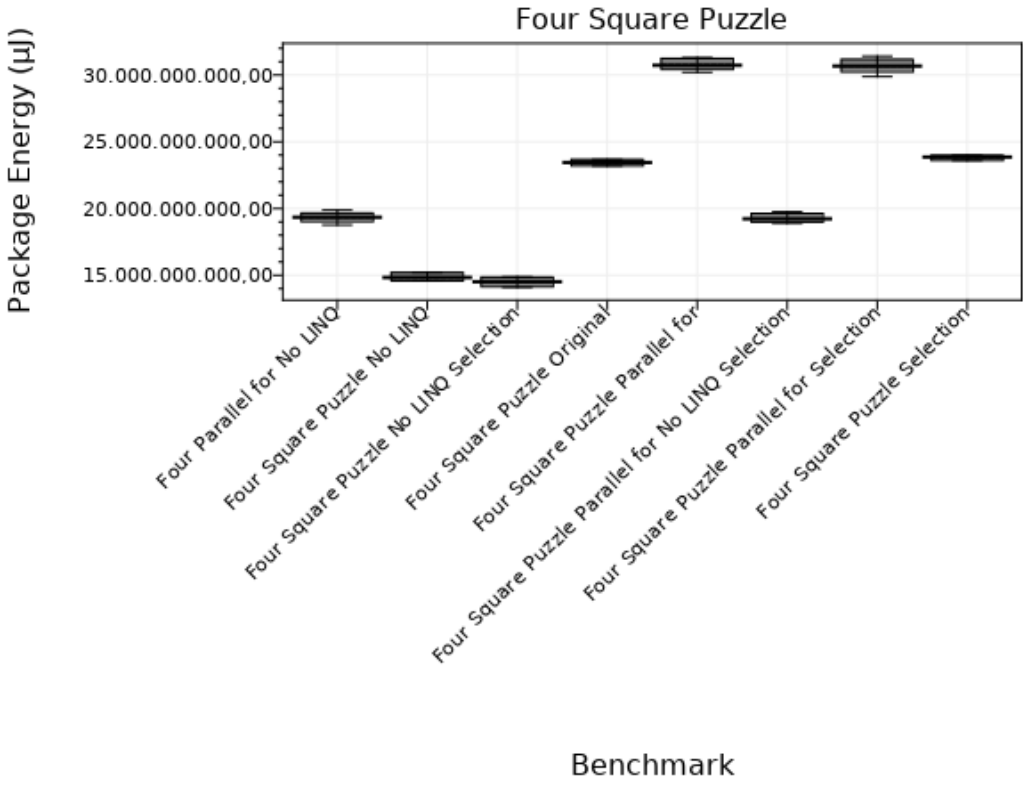

Figure 5.6: The energy consumption of Four Square Puzzle. The y-axis starts at 13.000.000.000 μJ and ends at 32.000.000.000 μJ.

In Figure 5.6 we see the energy consumption of the eight different variations of `Four Squares Puzzle`. We can see that the versions with LINQ changes have lower energy consumption than the original benchmark. The effects of `selection` changes appear negligible, while `concurrency` increases energy consumption. To see the variance observed for the benchmarks see Section A.6, Table A.95. The Kolmogorov-Smirnov tests are seen in Section A.9, Table A.126

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Change from Original
Four Parallel for No LINQ	741.746,01	19.344.332.730,75	474.296.726,87	- 17,49%
Four Square Puzzle No LINQ	1.120.423,58	14.812.726.562,50	625.608.685,66	- 36,82%
Four Square Puzzle No LINQ Selection	1.122.436,99	14.527.256.059,70	644.789.262,821	- 38,04%
Four Square Puzzle Original	1.786.165,078125	23.446.142.187,500	1.001.879.687,500	Not Applicable
Four Square Puzzle Parallel for	1.137.787,013	30.749.800.644,19	653.298.691,06	+ 31,15%
Four Square Puzzle Parallel for No LINQ Selection	739.534,97	19.233.176.847,68	466.418.699,67	- 17,97%
Four Square Puzzle Parallel for Selection	1.139.362,98	30.680.783.145,34	699.856.173,72	+ 30,86%
Four Square Puzzle Selection	1.806.450,82	23.836.241.796,88	1.009.103.906,25	+ 1,66%

Table 5.6: Table showing the elapsed time and energy measurement for each Four Square Puzzle.

In Table 5.6 we see the numerical values of the results. The p-values are seen in Section A.3.6.

Looking at the numeric results we see an increase in energy consumption of 31,15% by introducing `Parallel.For` in the original benchmark, however, the elapsed time is lower than the original. Removing LINQ usage decreases energy consumption by 36,82% while giving approximately the same speed boost as `Parallel.For`. Swapping all possible `if` statements for `switch` statements in the `selection` benchmark yields a small increase in energy usage of 1,66%. If we look at this in combination with the results of the Kolmogorov-Smirnov tests we see that `selection` is likely from a different distribution than the original, however:

- Parallel For & Parallel For Selection
- No LINQ & No LINQ Selection
- Parallel For No LINQ & Parallel For No LINQ Selection

are likely to be from the same distributions, and `selection` is thus unlikely to make a real difference.

We now check the plausibility of these results by performing sanity checks. We first check if the measurement from the original benchmark is plausible.

$$1.786.165,08 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 28.274.993.186,72\mu\text{J} \tag{5.11}$$

Equation 5.11 shows our estimated energy consumption for the original version of `Four Squares Puzzle`. This value and the one by the benchmark have a difference of 18,67%, which is a plausible difference.

We perform a second sanity check, on the benchmark with the lowest energy consumption.

$$1.122.436,99 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 17.768.177.618,04 \mu\text{J} \tag{5.12}$$

For the estimate of `Four Squares Puzzle No LINQ Selection` and the recorded result, we have a difference of 20,07%, which is a plausible difference.

Findings

The pattern in this section is that the removal of LINQ decreases energy consumption in all combinations it occurs in, while `Parallel.For` increases energy usage in all combinations but also results in the lowest runtime.

- Loop changes in the form of removing LINQ lead to a significant reduction in the energy consumption of the benchmark.
- Concurrency in the form of `Parallel.For` significantly increases energy consumption, while resulting in lower runtimes, which is not entirely unexpected given its performance on smaller workloads.
- Selection changes by converting to `switch` statements result in negligible changes in energy efficiency, contrary to what the previous results would suggest [3].

5.2.7 Numbrix Puzzle

The benchmark [75] is an implementation of a `Numbrix puzzle`, which is a grid of numbers, where one has to place a natural number in each blank square so that the sequence of numbered squares from "1" upwards can be marked by a line, where the line can only go to the immediate vicinity of the current square. We look at changing `Concurrency`, `Loops`, and `Selection` from Section A.10.

The following changes are made to the original benchmark. Instead of writing the results to the console, the strings are appended to a `StringBuilder`, and the length of the string is then returned from the benchmark. These changes are seen in our GitLab repository [1].

Concurrency

Regarding concurrency, we make changes such that the board is parsed concurrently using a `Parallel.For` loop, instead of a `for` loop. These changes are enacted in the two `Parse` methods where each row in the board is parsed concurrently.

Loops

LINQ has been substituted with a corresponding `for` loop, which tries to solve the puzzle, this is accomplished in one of the `Scan` methods. Another change here is in the `AreNeighbors` method, which has been changed from using `=>` for expression body definition, to instead making use of brackets and exchanging the use of LINQ with a `for` loop.

Selection

Regarding selection we change an `if` statement into an equivalent `switch` statement in one of the `Solve` methods.

Results

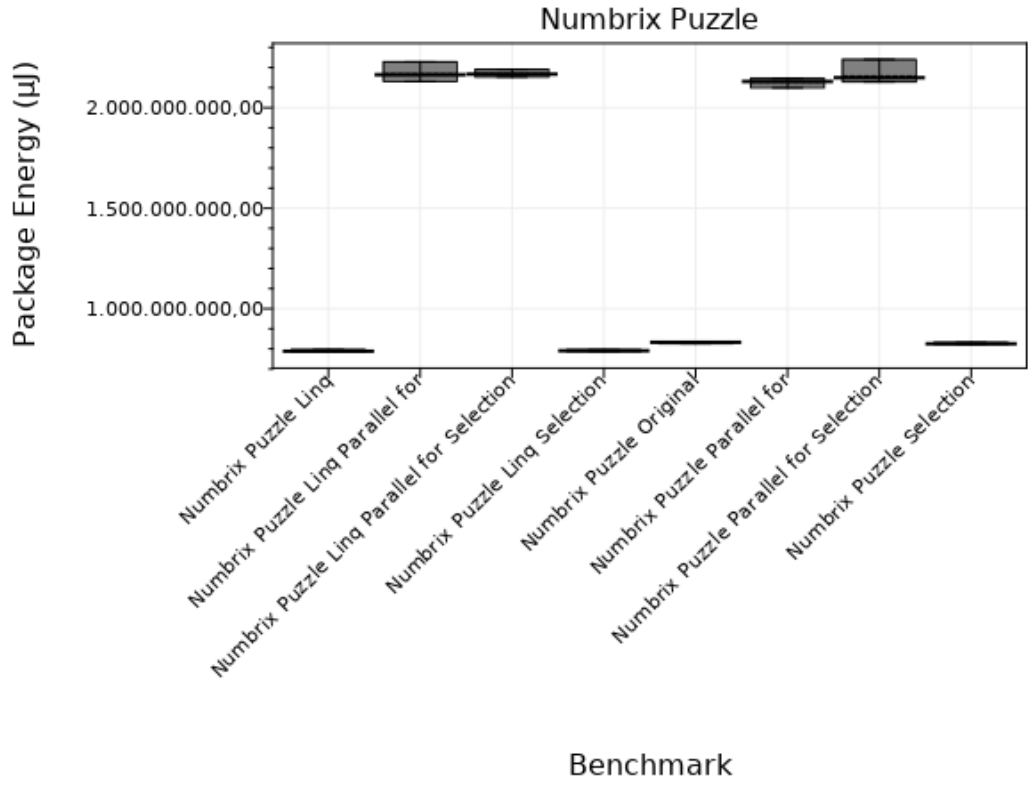

Figure 5.7: The energy consumption of Numbrix Puzzle. The y-axis starts at 700.000.000 μJ and ends at 2.300.000.000 μJ.

We see in Figure 5.7 how the use of concurrency in this benchmark significantly increases the energy consumption. We observe how the other changes are around the same level of energy consumption as the original benchmark. The variance observed for the benchmarks is seen in Section A.6, Table A.96 and the Kolmogorov-Smirnov tests are seen in Section A.9, Table A.127.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Change from Original
Numbrix Puzzle Linq	64.939,05	789.008.496,09	36.340.844,73	- 5,20%
Numbrix Puzzle Linq Parallel for	83.550,73	2.169.799.013,67	46.947.480,47	+ 160,69%
Numbrix Puzzle Linq Parallel for Selection	83.527,83	2.166.614.331,06	46.978.784,18	+ 160,31%
Numbrix Puzzle Linq Selection	65.168,12	790.740.014,65	36.467.504,88	- 5,00%
Numbrix Puzzle Original	68.344,20	832.330.566,40	38.240.747,07	Not Applicable
Numbrix Puzzle Parallel for	87.930,15	2.129.312.233,67	49.424.316,41	+ 155,83%
Numbrix Puzzle Parallel for Selection	87.711,81	2.157.190.162,30	49.324.834,91	+ 159,175
Numbrix Puzzle Selection	68.090,28	826.034.814,45	38.090.258,789	+ 0,76%

Table 5.7: Table showing the elapsed time and energy measurement for each Numbrix Puzzle.

In Table 5.7 we see the numerical values of the results. The p-values are seen in Section A.3.7. We see that all changes except for concurrency-related ones lead to a decreased energy consumption. This is not unexpected as the results from Section 4.1.1 shows that using `Parallel.For` requires a substantial workload before the approach is preferable to sequential execution. While the effect is not as large as observed in other benchmarks, we find that removal of LINQ has the largest impact on improving the energy consumption of the benchmark. If we only look at the numbers in Table 5.7 it looks like combining the removal of LINQ with selection changes lessens the impact, however, if we look at the variance table we find that on average the energy consumption is improved when these changes are combined. If we look at these results in conjunction with the Kolmogorov-Smirnov tests, we find for the original version and `Selection` that they are probably part of the same probability distribution, which fit with the change shown in Table 5.7. According to these tests, the LINQ version and `Linq Selection` version are likely part of the same distribution. This fits with the idea that selection does not carry a significant impact. As such it leads to the conclusion that in this particular benchmark `Selection` has a negligible effect and any improvements seen when combined with LINQ are likely part of the variance exhibited by LINQ. We also have from these tests that all the versions of the benchmark which involve concurrency are probably part of the same probability distribution, which leads us to the conclusion that the increased energy consumption is solely from `Parallel.For`.

Next we look at plausibility of these result using sanity checks. We check if the measurement from the original benchmark is plausible.

$$68.344,20 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 1.081.888.686,00\mu\text{J} \tag{5.13}$$

We find a difference of 26,07% between estimated and measured energy consumption, which is plausible.

Next we perform a sanity check on the benchmark with the lowest energy consumption.

$$64.939,05 \text{ ms} \cdot 1000 \cdot 15,83 \text{ Watts} = 1.027.985.161,50\mu\text{J} \tag{5.14}$$

For the estimated value of `Numbrix Puzzle Linq` and the recorded value, there is a difference of 26,30%, which is plausible.

Findings

The results here show us that removing LINQ positively impacts energy consumption, and continues to do so when combined with `switch`. The changes contributed by these two however have a negligible effect when used in conjunction with `Parallel.For`. The low performance of `Parallel.For` might be caused by the size of the particular workload, as found in Section 4.1.1.

- Concurrency results in increased energy consumption and execution time, which is not unexpected given the limited workload in this benchmark.
- Loop changes, where LINQ is substituted for equivalent `for` loops results in a non-negligible reduction to energy consumption.
- Selection change to `switch` has a negligible effect, contrary to earlier findings [3].
  - This effect can increase when combined with changes to `Loops` in the form of LINQ removal according to the variance table results. However, according to the Kolmogorov-Smirnov tests, these are likely part of the same distribution, which means that the improved energy consumption is likely due to variance.

5.2.8 Sum to Hundred

The benchmark [76] is an implementation of the `Sum To Hundred` benchmark, where one has to use the mathematical operators plus or minus before any of the digits in the numeric string "123456789" such that the resulting mathematical expression adds up to 100. We look at changing `Objects`, `Selection`, and `Loops` from Section A.10.

A few changes are made to this benchmark. First, the console output stream is set to a null stream to avoid printing to the console. Furthermore, instead of printing the results the number of solutions to the 100th result is returned from the benchmark.

Objects

The benchmark contains a class called `Expression` which is heavily used. This is refactored into a `struct` as it reduces the cost of calling its methods, according to the information presented in [3].

Selection

For `Selection` we exchange `if` statements with equivalent `switch` statements where possible.

Loops

The original benchmark contains two parallel queries that create expressions. These queries are replaced with a `Parallel.For` loop that instead creates these expressions and stores them in an array.

Results

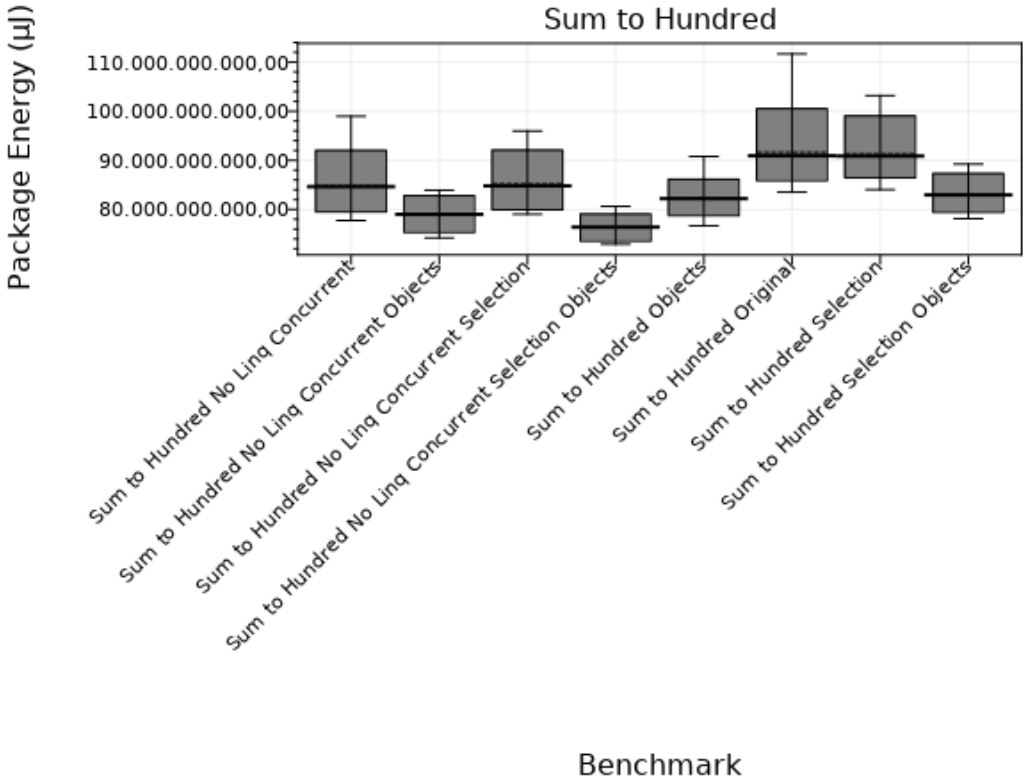

Figure 5.8: The energy consumption of Sum To Hundred. The y-axis starts at 72.000.000.000 μJ and ends at 114.000.000.000 μJ.

In Figure 5.8 we see the results of running the `Sum To Hundred` benchmark. We see that using a `struct` instead of a `class` reduces energy consumption, but also that swapping the use of parallel LINQ for `Parallel.For` decreases energy consumption. This is the case for all changes, as well as when they are combined.

To see the variance observed for the benchmarks see Section A.6, Table A.97. We subject the results to Kolmogorov-Smirnov tests, which are seen in Section A.9, Table A.128.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)	Change from Original
Sum to Hundred No Linq Concurrent	5.899.848,63	84.788.018.251,81	5.528.268.727,36	- 7,49%
Sum to Hundred No Linq Concurrent Objects	5.395.324,07	78.937.738.415,24	4.712.789.783,97	- 13,87%
Sum to Hundred No Linq Concurrent Selection	5.842.068,42	85.191.762.348,40	5.518.709.678,21	- 7,05%
Sum to Hundred No Linq Concurrent Selection Objects	5.111.291,75	76.287.187.411,72	4.495.824.682,20	- 16,76%
Sum to Hundred Objects	4.952.060,02	82.198.099.272,63	4.206.246.060,08	- 10,31%
Sum to Hundred Original	5.780.446,61	91.651.333.850,77	5.215.095.645,87	Not Applicable
Sum to Hundred Selection	5.690.152,70	91.313.060.486,58	5.124.518.036,91	- 0,37%
Sum to Hundred Selection Objects	4.959.930,67	82.962.779.845,03	4.220.242.293,94	- 9,48%

Table 5.8: Table showing the elapsed time and energy measurement for each Sum to Hundred.

In Table 5.8 we see the numerical values of the results. The p-values are seen in Section A.3.8. We see how each of the benchmarks shows a reduction in energy consumption. Additionally, we see that the benchmark which consumed the least amount of energy was the one combining all three types of changes. This resulted in an energy consumption decrease of 16,76%.

We now check the plausibility of these results by performing our sanity check. We first check if the measurement from the original benchmark is plausible.

$$5.780.446,61 \text{ ms} \cdot 1000 \cdot 4 \text{ cores} \cdot 15,83 \text{ Watts} = 366.017.879.345,20 \mu\text{J} \quad (5.15)$$

The result of Equation 5.15 is a 119,90% difference compared to the original benchmark. This is deemed plausible because the estimate does not take into account that not all cores perform the same amount of work. We do a second check with the benchmark with the lowest package energy consumption.

$$5.111.291,75 \text{ ms} \cdot 1000 \cdot 4 \text{ cores} \cdot 15,83 \text{ Watts} = 323.646.993.610,00 \mu\text{J} \quad (5.16)$$

The resulting energy consumption of Equation 5.16 and the benchmark end up having a difference of 123,70%. We once more deem this plausible, even though it exceeds the pre-established cutoff value, because it does not take into account that not all cores perform the same amount of work, nor are they always active over the course of the benchmark.

Findings

The results show us that combining the different changes in a single benchmark results in the largest reduction in energy consumption. In contrast

to the Dijkstra benchmark, we see that this benchmark has a decrease in energy consumption when performing a change to `Objects`.

- The benchmark combining the types of changes resulted in the largest decrease in energy consumption.
- Changing a class to a `struct` decreased the energy consumption by 10,31%
- `Selection` showed a negligible change when used alone, which is different from what the findings in [3] suggest the effect would be.

5.2.9 Summary

Not all changes to benchmarks have the expected outcomes. Some changes have no effect or a different effect than previous findings suggest. Some changes that have a positive or negligible effect on their own make the benchmark consume more energy when combined with other changes.

These are the surprising results categorized under their respective benchmarks:

- `100 Prisoners`
  - Collection changes do not significantly reduce energy consumption when used together with Concurrency.
- `Almost Prime`
  - Concurrency leads to both increased elapsed time and energy consumption.
  - Collection changes do not significantly change energy consumption.
  - Object changes do not reduce energy consumption.
- `Chebyshev Coefficients`
  - Concurrency reduces elapsed time and increases energy consumption.
  - Datatype and Collections changes together result in a reduction of the positive effects of using an array.

- `Compare length of two strings`
  - Loops changes in the form of changing `for` to `foreach` increases energy consumption.
  - Selection changes in the form of swapping `if` statements for `switch` statements has negligible effect.
- `Dijksta's Algorithm`
  - Combining changes to `Objects` and `Collections`, leads to an increased energy consumption, where the two individual changes had no significant impact.
- `Four Squares Puzzle`
  - Selection changes where `if` statements are swapped for `switch` statements has negligible effect.
- `Numbrix Puzzle`
  - Selection changes where `if` statements are swapped for `switch` statements has negligible effect.

`Sum to Hundred`

  - `Selection` shows a negligible change on its own.

These surprising results serve as the subjects for the analysis of the macrobenchmark analysis.

5.3 Result Analysis

This section analyzes the results gathered in Section 5.2. We do this to determine the reason for behavior that was observed to be different than in Section 4.1. It also allows us to determine why certain changes should or should not be combined.

The process for this analysis is as follows:

- Look at the benchmark, to determine if two or more changes have an impact on one another.
- Check if findings in Section 4.2 explain the results.

- Look at the documentation for the language constructs.
- Create new benchmarks to test unexplored uses of language constructs.

5.3.1 100 Prisoners

For the 100 Prisoners benchmark, we have observed a single result that is unexpected, this is:

- Collection change does not significantly reduce energy consumption when combined with Concurrency.

Concurrency and Collections

The results in Table 5.1 show that only changing `Collections` carries a significant impact on energy consumption, which is the case across 10 runs. However when we look at exchanging `List` for array in conjunction with `Concurrency` we find that there is no significant impact, according to our Kolmogorov-Smirnov test, Section A.9.1. However, using the information presented in Table A.90, we find that the mean energy consumption shows a 0,60% increase, when comparing `Prisoners Parallel for Array` to `Prisoners Parallel for`. We also see that in 6 out of 10 cases that using an array results in higher energy consumption. Taking these facts into account and the variance exhibited by the two benchmarks, we conclude that the effect on energy consumption is too small to be measured as significant by the Kolmogorov-Smirnov tests.

5.3.2 Almost Prime

For the `Almost Prime` benchmark the surprising observations are:

- Concurrency leads to both increased elapsed time and energy consumption.
- Collection change do not change energy consumption.
- Object changes do not change energy consumption.

Concurrency

In the benchmark, there is a value called `K` which describes the number of primes that constitute a given number. This `K` is tried with values ranging from 1 to 10, and a larger `K` gives a larger amount of work in the loop that has been made parallel.

We hypothesize that the computational work is too low to benefit from concurrency, and thus the benchmark is run with a larger value of `K`.

The benchmark is modified to run with `K` being 20 and 25 to confirm if the small values of `K` induce the overhead. The reason for choosing 25 instead of 30 is that the workload in this benchmark increases exponentially and will take too long if using 30.

Package Energy (μJ)	Original	ManualThread
Run 1	52.328.501.515.151,50	59.339.908.471.830,90
Run 2	56.631.788.600.000,00	55.985.753.734.375,00
Run 3	54.889.269.900.000,00	56.708.564.890.804,60
Run 4	53.454.128.473.684,20	59.634.143.420.689,60
Run 5	51.892.349.975.609,70	56.736.578.204.819,20
Run 6	52.423.984.547.619,00	56.649.109.538.681,90
Run 7	56.572.011.000.000,00	56.367.450.406.565,60
Run 8	56.602.034.000.000,00	56.124.369.986.702,10
Run 9	58.950.644.500.000,00	59.451.917.141.447,30
Run 10	58.994.648.175.438,50	56.416.153.077.994,40
Mean	55.273.936.068.750,30	57.341.394.887.391,10
Percentage variance	12,04%	6,12%

Elapsed Time (ms)	Original	ManualThread
Run 1	4.610.258.066,67	1.832.485.804,58
Run 2	5.129.702.870,00	1.740.104.351,04
Run 3	4.607.189.910,00	1.751.989.952,59
Run 4	4.606.749.705,26	1.834.834.188,28
Run 5	4.606.814.334,15	1.765.328.323,37
Run 6	4.606.580.933,33	1.757.368.903,15
Run 7	5.130.773.960,00	1.753.616.620,96
Run 8	5.130.151.320,00	1.740.170.564,89
Run 9	5.130.191.425,00	1.825.339.622,70
Run 10	5.130.891.991,23	1.744.571.126,18
Mean	4.868.930.451,56	1.774.580.945,77
Percentage variance	10,22%	5,16%

Table 5.9: Results of running the Almost Prime with `K`=20 benchmarks 10 times and percentage variance between highest and lowest. Recorded energy consumption to the left and elapsed time to the right.

In Table 5.9 we see the numeric results of running the `Almost Prime` benchmark with a value of `K` at 20. The results show that the mean energy consumption of the concurrent benchmark is 3,67% higher, which is significantly less than the difference seen in Section 5.2.2.

To see if this trend continues and the concurrent version becomes more efficient, another benchmark is run where `K` is 25.

Package Energy (μJ)	Original	ManualThread
Run 1	8.079.962.912.000.000,00	6.450.289.912.200.000,00
Run 2	7.015.695.260.200.000,00	5.856.756.225.294.110,00
Run 3	7.256.377.395.400.000,00	5.858.823.502.294.110,00
Run 4	7.025.564.949.200.000,00	5.787.913.131.000.000,00
Run 5	8.184.599.836.000.000,00	5.813.425.547.294.110,00
Run 6	7.244.094.797.200.000,00	5.870.727.754.411.760,00
Run 7	7.759.847.124.000.000,00	5.817.930.353.687.500,00
Run 8	7.155.172.466.400.000,00	5.835.808.977.529.410,00
Run 9	7.754.970.640.750.000,00	5.840.111.133.529.410,00
Run 10	7.785.266.080.250.000,00	5.819.931.576.764.700,00
Mean	7.526.155.146.140.000,00	5.895.171.811.400.510,00
Percentage variance	14,28%	10,27%

Elapsed Time (ms)	Original	ManualThread
Run 1	699.262.017.175,00	184.833.484.506,67
Run 2	623.203.333.460,00	166.444.851.870,59
Run 3	623.225.630.940,00	166.539.790.605,88
Run 4	623.238.975.020,00	164.553.938.740,00
Run 5	699.341.901.275,00	165.259.670.717,65
Run 6	623.207.673.160,00	166.631.590.223,53
Run 7	699.287.267.150,00	165.502.113.406,25
Run 8	623.204.505.460,00	165.707.331.594,12
Run 9	699.274.847.325,00	166.421.354.070,59
Run 10	699.281.268.675,00	165.517.499.223,53
Mean	661.252.741.964,00	167.741.162.495,88
Percentage variance	10,89%	10,97%

Table 5.10: Results of running the Almost Prime with `K`=25 benchmarks 10 times and percentage variance between highest and lowest. Recorded energy consumption to the left and elapsed time to the right.

The results of running with `K` being 25 is seen in Table 5.10. We see the concurrent version using less power than the original, and being approximately four times faster. Based on these trends we conclude the small workload is the cause of the original higher energy usage of the concurrent benchmark.

Collections

Upon further analysis of the benchmark, the collection or array is only used to store 10 integers, which are accessed once each. The energy spent on accessing these elements is thus insignificant to the total amount of work in the benchmark and one should not expect to see any significant changes.

Objects

The class `KPrime` has been changed to a `struct`. In the benchmark, however, `KPrime` is only instantiated once, and a single method is called once. This again means that the total overhead of using `KPrime` is insignificant compared to the total work and no significant changes should be expected.

5.3.3 Chebyshev Coefficients

For the Chebyshev Coefficients benchmark the surprising results are:

- Concurrency reduces elapsed time and increases energy consumption.
- Datatype and Collections together reduce the positive effects of Collection changes.

Concurrency

While the use of concurrency reduces the elapsed time of the benchmark it also increases the energy consumption. The benchmark contains a value called `nX` which is used to determine the number of approximations performed in the benchmark. This number has already been increased from 20 to 100.000.000, which constitutes a larger workload. Part of the changes to the original version is saving all results across iterations within the benchmark, instead of utilizing only the result of the very last iteration. We suspect that this might be the cause of the higher energy consumption. Therefore we create another variation of the original benchmark, which saves the result of each iteration in an array.

Package Energy (μJ)	Array Storing	Original	Concurrent
Run 1	68.652.412.200.000,00	49.178.185.105.263,10	59.392.086.333.333,30
Run 2	69.098.047.100.000,00	51.310.030.900.000,00	60.420.758.440.000,00
Run 3	68.642.915.100.000,00	46.769.783.900.000,00	60.046.398.666.666,60
Run 4	68.498.860.200.000,00	47.315.497.800.000,00	59.202.134.250.000,00
Run 5	68.246.907.900.000,00	50.720.673.400.000,00	59.790.862.500.000,00
Run 6	68.845.233.900.000,00	48.675.994.875.000,00	59.820.827.235.294,10
Run 7	68.570.082.000.000,00	46.945.351.000.000,00	58.992.860.600.000,00
Run 8	68.496.290.800.000,00	49.113.954.545.454,50	59.950.220.933.333,30
Run 9	68.703.779.100.000,00	47.777.393.125.000,00	60.167.001.000.000,00
Run 10	69.178.619.300.000,00	47.535.673.076.923,00	59.653.132.000.000,00
Mean	68.693.314.760.000,00	48.534.253.772.764,10	59.743.628.195.862,70
Percentage variance	1,35%	8,85%	2,36%

Table 5.11: Results of running the Array storing benchmark with results stored in an array 10 times and percentage variance between highest and lowest recorded energy consumption, compared to the original and concurrent version of Chebyshev Coefficients.

We can see in Table 5.11 the results of the Array storing benchmark compared to the original and concurrent version. We calculate that the storing of results in an array increases the energy consumption by 41,54%, this is around twice as much as the increase from Original to Concurrent. If we then compare the Concurrent and Array Storing versions, we find that the energy consumption is decreased by 13,03%. We conclude from this that the increase in energy consumption from the original Chebyshev Coefficients to the Concurrent version is not caused by the concurrency itself but by the synchronization of results needed to return the same value as the original.

Datatype and Collections

The results in Section 5.2.3 show that the Datatype changes results in slightly higher energy consumption across all 10 runs, see Section A.6, Table A.92, this difference is not evaluated to be significant according to the Kolmogorov-Smirnov tests in Section A.9, Table A.123. If we look at the mean of `Array` and `Datatype` and find the mean of these two values, we end up with an energy consumption of 47.552.593.964.336,10 μJ. If we compare this to the mean of `ArrayDatatype`, we find a difference between these of 1,31%. Taking the variance of these different benchmarks into account, it is plausible that the decrease in energy efficiency observed for `ArrayDatatype` compared to `Array` comes from the small increase in energy consumption when enacting the `Datatype` change.

5.3.4 Compare length of two strings

This section analyzes the surprising results from the `Compare length of two strings` benchmark, which are:

- Loops changed to `foreach` increases energy consumption.
- Selection changed to `switch` statements have negligible effect.

Loops

For `Loops`, we exchange two `for` loops for `foreach` loops.

```
1  //Before
2  for (int i = 0; i < strings.Length; i++)
3      li[i] = (strings[i].Length, i);
4  ...
5  for (int i = 0; i < strings.Length; i++) {
6      int length = li[i].Item1;
7      string str = strings[li[i].Item2];
8      ...
9  }
10
11 //After
12 foreach (var element in strings) {
```

```
13      li[Array.IndexOf(strings, element)] = (element.Length,
        ↪  Array.IndexOf(strings, element));
14  }
15  ...
16  foreach (var element in strings) {
17      int length = li[Array.IndexOf(strings, element)].Item1;
18      string str = strings[li[Array.IndexOf(strings,
        ↪  element)].Item2];
19      ...
20  }
```

Listing 42: The changes to Loops in `Compare length of two strings`.

Listing 42 shows the changes made to the benchmark to facilitate the use of `foreach`. We see how an extra operation is added to the loop to keep track of the index of the array. We use `Array.IndexOf` [77] instead of the index of the regular `for` loop. The `Array.IndexOf` method searches through every element in a one-dimensional array until it finds the correct element. We can conclude from this that using `Array.IndexOf` is the cause of the increased energy consumption, compared to the original version.

Selection

For `Selection` we change an `if else if` with a `switch` statement.

```
1   //Before
2   if (length == maxLength)
3       predicate = predicateMax;
4   else if (length == minLength)
5       predicate = predicateMin;
6   else
7       predicate = predicateAve;
8
9   //After
10  switch (length) {
11      case var _ when length == maxLength:
12          predicate = predicateMax;
13          break;
14      case var _ when length == minLength:
```

```
15          predicate = predicateMin;
16          break;
17      default:
18          predicate = predicateAve;
19          break;
20  }
```

Listing 43: The changes to Selection in `Compare length of two strings`.

When we look at Listing 43, we can see that the switch statements utilize the `var` pattern [78]. This type of `switch` has not been tested, and therefore we do not know its impact on energy consumption. To test how using this pattern compares to the `switch` statements tested in [3], we construct an experiment that compares this type of `switch` statement to the different `if` statements explored in [3, p.64-76].

Package Energy (μJ)	If	IfElseIf	IfElse	VarPattern
Run 1	9.845,95	10.513,12	10.596,52	9.908,42
Run 2	10.735,73	10.283,18	10.611,20	10.266,79
Run 3	10.640,25	10.405,54	10.286,44	10.590,27
Run 4	10.672,76	10.610,03	10.672,83	10.426,51
Run 5	10.855,88	11.049,21	10.725,51	10.746,92
Run 6	10.743,57	10.969,57	11.022,26	10.992,06
Run 7	11.065,61	10.636,12	10.873,85	10.890,29
Run 8	10.756,67	11.197,48	10.826,99	10.919,88
Run 9	11.230,69	10.910,64	11.059,29	10.871,89
Run 10	10.974,02	10.942,84	11.027,11	11.348,36
Mean	10.752,11	10.751,77	10.770,20	10.696,14
Percentage variance	12,33%	8,17%	6,99%	12,69%

Table 5.12: Results of running the Selection benchmarks 10 times and percentage variance between highest and lowest recorded energy consumption.

In Table 5.12 we see the results of running the `if` benchmarks from [3] with the addition of the new `VarPattern` benchmark. We calculate that the percentage difference between the mean of `VarPattern` and `IfElse` is 0,69%. This is a minor difference, given that these two benchmarks have the largest difference in mean energy consumption in Table 5.12. This leads us to suspect that the energy impact of these different approaches to selection is essentially the same. To test this we conduct Kolmogorov-Smirnov tests.

Package Energy p-values	If	IfElse	IfElseIf	VarPattern
If	_	0,962	0,309	0,664
IfElse	0,962	_	0,962	0,664
IfElseIf	0,309	0,962	_	0,962
VarPattern	0,664	0,664	0,962	_

Table 5.13: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of selection statements with regard to Package Energy.

In Table 5.13 we see the results of the Kolmogorov-Smirnov tests. We see that all of the results are likely to be from the same distribution, and there is thus no significant difference between `var` pattern and a normal `if` statement. This explains why changing selection statements has a negligible effect.

5.3.5 Dijkstra's Algorithm

In this section, we analyze the surprising finding for the `Dijkstra` benchmarks, which are:

- Combining changes to `Objects` and `Collections`, leads to an increased energy consumption.

Collections and Objects

The combination of `Array` and `Objects` resulted in an increase of energy consumption of 2,69% whereas the individual changes were negligible. Using the mean in the variance table seen in Table A.94, we calculate the difference in energy consumption from the original benchmark. We then compare the values of these differences to check if the individual changes of `Array` and `Objects` add up to the same amount as the `ArrayObjects` benchmark.

This results in the values 495.520,64 μJ and 954.394,10 μJ for the `Array` and `Objects`. For the `ArrayObjects` benchmark there is a difference of 1.715.509,59 μJ. Adding the differences between the two individual benchmarks we get 1.449.914,74 μJ. As such the two individual changes add up to a result that is 16,7% different from the `ArrayObjects` benchmark.

Based on this calculation we conclude that the energy consumption of `ArrayObjects` is the accumulation of the energy consumption changes in the two individual benchmarks given the variance seen in Table A.94. Additionally we see in the Kolmogorov-Smirnov tests in Table A.125 that the

ArrayObjects benchmark is likely not part of a different distribution than the Array and Objects benchmarks.

5.3.6 Four Squares Puzzle

In this section, we analyze the surprising finding for the Four Squares Puzzle benchmark, which is:

- Selection changes by converting to switch statements result in negligible changes.

Selection

For selection, if statements are replaced by a switch statement.

```
1  //Before
2  if (notValid(unique, d, c, b, a)) continue;
3  if (fp != b + c + d) continue;
4
5  //After
6  switch (unique) {
7          case var _ when notValid(unique, d, c, b, a):
8                  continue;
9          case var _ when (fp != b + c + d):
10                 continue;
11 }
```

Listing 44: The changes to selection statements in Four Square Puzzle.

In Listing 44 we see the changes made to the benchmark. The switch statement is not one of the normal switch statements that we tested in [3] which makes a jump table based on the value of a variable, in this case, the unique variable. What is used here is called a var pattern [78]. It is used as the if statements can not be converted to a standard switch statement.

The var pattern is explored in Section 5.3.4, where it was found that it was not significantly different from the different if statements. This also explains what is observed with the Four Squares Puzzle benchmark, so we can conclude that the reason for switch not being different from the original, is because using if or switch with the var pattern makes no significant difference in energy consumption.

5.3.7 Numbrix Puzzle

We analyze the reason behind the surprising finding for Numbrix Puzzle which is:

- Selection changes to `switch` statements have negligible effect.

Selection

We here look at the changes to an `if else` chain. And why it does not have the expected impact according to earlier findings.

```
1  //Before
2  if (x < 0 || x >= size.h || y < 0 || y >= size.w) return false;
3  if (board[x, y] < 0) return false;
4  if (given[n - 1]) {
5      if (board[x, y] != n) return false;
6  }
7  else if (board[x, y] > 0) return false;
8
9  //After
10 switch (x) {
11     case var _ when x < 0 || x >= size.h || y < 0 || y >=
       ↪  size.w:
12         return false;
13     case var _ when board[x, y] < 0:
14         return false;
15     case var _ when given[n - 1]:
16         if (board[x, y] != n) {
17             return false;
18         }
19         break;
20     case var _ when board[x, y] > 0:
21         return false;
22 }
```

Listing 45: The changes to Selection in `Numbrix Puzzle`.

Looking at the code in Listing 45, we see that to change from using `if` and `else` to `switch` we have to use the `var` pattern [78], as there is no constant value which can be used for the `switch`.

This pattern is explored in Section 5.3.4. It was found that the `var` pattern was not significantly different from `if` statements regarding energy consumption. This explains the result for Selection in the `Numbrix Puzzle` benchmark. Using the `var` pattern with `switch` is not significantly different from `if` in regard to energy consumption, which means that the similar values between the original and selection version of the benchmark make sense.

5.3.8 Sum to Hundred

The benchmark `Sum To Hundred` has a single surprising finding which is:

- `Selection` shows negligible change.

Selection

In the `Sum To Hundred` benchmark three `if` statements are replaced by `switch` statements.

```
1  //Before
2  if (times == 0) {
3      Gaps[i] = Operations.Join;
4  }
5  else if (times == 1) {
6      Gaps[i] = Operations.Minus;
7  }
8  else {
9      Gaps[i] = Operations.Plus;
10 }
11
12 //After
13 switch (times) {
14     case 0:
15         Gaps[i] = Operations.Join;
16         break;
17     case 1:
```

```
18          Gaps[i] = Operations.Minus;
19          break;
20      default:
21          Gaps[i] = Operations.Plus;
22          break;
23  }
```

Listing 46: The changes to selection in `Sum To Hundred`.

In Listing 46 an example change can be seen. As can be seen here, we switch on `times`, which is an `int`, and there are two specific cases and a default case. The other two refactored statements switch on the value of an `enum` and have one specific case and a default case.

Considering that the `switch` statements checked in [3] involve multiple cases and do not utilize `enum`, the `switch` statements used in `Sum To Hundred` have not been explored prior to this. As such, benchmarks that compare a `switch` statement with an `if` statement in these circumstances are created. Also created are benchmarks for testing `switch` and `if` on an `enum` where there is a single explicit case and a default case.

Package Energy (μJ)	IfTwoCase	SwitchTwoCase
Run 1	942.457,53	944.244,21
Run 2	959.889,25	949.146,25
Run 3	958.621,14	959.486,29
Run 4	979.423,02	982.635,99
Run 5	967.860,99	971.192,92
Run 6	975.729,75	982.155,84
Run 7	995.563,64	993.048,97
Run 8	944.987,61	956.650,35
Run 9	985.954,12	996.092,91
Run 10	933.952,62	943.604,03
Mean	964.443,97	967.825,78
Percentage variance	6,19%	5,27%

Package Energy (μJ)	IfEnum	SwitchEnum
Run 1	17.101,02	17.205,72
Run 2	19.771,64	17.708,35
Run 3	17.955,59	18.076,60
Run 4	19.905,55	19.967,69
Run 5	18.654,85	19.930,86
Run 6	20.453,33	19.955,76
Run 7	20.572,57	19.975,09
Run 8	20.400,01	19.810,04
Run 9	17.855,28	19.925,31
Run 10	19.794,69	17.971,77
Mean	19.246,45	19.052,72
Percentage variance	16,87%	13,86%

Table 5.14: Results of running the `switch` and `if` benchmarks 10 times and percentage variance between highest and lowest. The benchmarks testing two cases and a default is on the left, and the benchmark using `enum` and one case and a default is on the right.

Looking at the values in Table 5.14 we see that the difference in mean values is 0,35% for `IfTwoCase` and `SwitchTwoCase`, while the difference between `IfEnum` and `SwitchEnum` is 1,01%. Given how close the mean values are to one another and the variance, we suspect that the different benchmarks are likely part of the same probability distribution for both comparisons. This is tested in two Kolmogorov-Smirnov tests.

Package Energy p-values	IfEnum	SwitchEnum
IfEnum	_	0,664
SwitchEnum	0,664	_

Package Energy p-values	IfTwoCase	SwitchTwoCase
IfTwoCase	_	0,962
SwitchTwoCase	0,962	_

Table 5.15: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of the benchmark comparing `if` and `switch` with regard to Package Energy.

The *p*-values of the Kolmogorov-Smirnov tests in Table 5.15 shows that for both sets of benchmarks the results from `if` and `switch` are most likely from the same distribution. We conclude that `switch` is not more energy-efficient than `if` under these conditions.

5.3.9 Summary

Through our analysis, we have delved into the surprising results of our macrobenchmarks. In the analysis, we have discovered the reasoning behind these results and found connections linking the resulting energy consumption and the code changes. In the analysis we have come to the following conclusions:

- Collections
  - Changing `List` into `Array` causes a small but negligible decrease in energy consumption, the cause of this was found to be the overall energy consumption of benchmarks, which meant that the small changes from this change were not deemed significant by the Kolmogorov-Smirnov tests.
- Concurrency
  - A small workload makes the use of concurrency ineffective and causes a large increase in energy consumption. This is because the overhead caused by managing concurrency outweighs the benefit when the workload is small.
  - Concurrency can also require changes to data stored to facilitate the return of the same data as the original version of a benchmark, which increases energy costs.
- Datatypes
  - Refactoring `int` to `uint` yields insignificant differences as the potential gains from these refactorings are so small compared to the overall energy usage of the benchmarks.

- Objects
  - Changing a class into a struct causes an insignificant change in energy consumption, the cause of this was found to be low usage of instantiation and method calls, where improvements from this change require that these are common operations.
- Selection
  - The use of selection has not significantly reduced energy consumption, either because a `var` pattern is used instead, or because not enough cases are present in the `switch` statement, but also because selection in our benchmarks constitutes a small amount of the total energy consumption.

Chapter 6

Reflections

We reflect on the benchmarks we have used, both concerning microbenchmarks and macrobenchmarks. The use of information gathered in [3], alongside the tool developed in that project is also part of this process. We discuss how our choice to use variance tables and Kolmogorov-Smirnov tests have influenced our process and findings. Finally, we look at the work process we have utilized and its effects on our project.

6.1 Choice of Benchmarks

The choice of manually making our concurrency microbenchmarks allowed us to test specifically what we intended. However, the validity of the benchmarks can be discussed as they do not cover all potential uses of the constructs. It should also be kept in mind that the benchmarks we have chosen are more or less directly parallelizable, which is not always the case. It could therefore be a benefit to look at benchmarks that are not fully parallelizable. Finding benchmarks from another source can potentially improve the validity, by increasing the sample size of our results, adding other potential use cases, and gathering results from different degrees of parallelizability. Larger concurrency problems like k-means clustering, raytracing, or Santa Claus problem are left for future work.

The workload chosen for the concurrency microbenchmarks can also be discussed, as the exact numbers do not have a deeper scientific meaning other than testing different orders of magnitudes of work, but have given useful results.

A thing that might skew the results is a large amount of `LoopIterations`

present in the different benchmarks to get a runtime of at least a quarter second. Due to this, `LoopIterations` can be over 500 million. This means some overhead is neglected, as running a loop with 10 iterations, and a small workload will yield significantly more relative overhead than running 500 million iterations with the same workload.

Using macrobenchmarks from *Rosetta* has enabled us to spend less time writing benchmarks, as they have predefined tasks. However, some of these benchmarks are already optimized by those who have written them, and optimizing further has proven a challenge, this is also positive as it allows us to better see the effects of language construct changes on code that is optimized. We have had to alter some benchmarks to reach a runtime larger than a few milliseconds, as such, not all benchmarks are as computationally intensive. This has provided us with knowledge about how the language constructs affect both computationally intensive and non-intensive work.

6.2 Use of Libraries/Previous Work

The use of the library from our previous work [3] has proven to work reliably for the microbenchmarks in this project, but also for the macrobenchmarks even though it was constructed mostly focusing on microbenchmarks. As such it has allowed us to focus on creating and analyzing the results of different benchmarks.

The use of the PowerUp [41] tool has allowed us to view release mode assembly for the benchmarks, and thus gain insight into the generated assembly during the analysis of the microbenchmarks.

6.3 Result Variance and Kolmogorov-Smirnov

During the creation and experimentation of our microbenchmarks, we found that individual benchmarks had varying results. To account for variance in runs, we increase the number of times each benchmark is tested. This results in variance tables containing 10 runs of a benchmark, and a calculated mean, which increases the validity of the measured results. This has allowed us to better determine if the observed results, were representative of benchmarks or if they were outliers. Because of this, we have been capable of drawing more reliable conclusions about the effect of language constructs than otherwise.

Because of the variance, it is also difficult to discern between benchmarks of similar energy consumption. To counter this we make use of Kolmogorov-Smirnov tests. These tests are used to compare two samples and tell us how likely these samples have been drawn from the same distribution. Thus supporting whether a change in energy consumption between benchmarks is significantly different.

6.4 Work Process

Our work process has helped us throughout the project. The hybrid approach we utilize has allowed for clearly established goals while also allowing for easy modification of our plans when challenges occurred.

The utilization of stand-up meetings has made it so that we have always had a clear view of what task each member of the project group is working with, as well as how far along a given task is. This has allowed us to establish a better overview of how far in the project we are. This has been supported by our use of a kanban board, which supports our knowledge of what tasks are underway, how far along they are, tasks that are yet to be done, and those that are finished.

We have also made use of `GitLab`, which has allowed for easy tracking of which benchmarks have been completed. The use of branches has allowed us to separate our microbenchmarks from our macrobenchmarks, which has helped keep them separate and keep the overview of what is completed or not. The choice of requiring approval for a merge before a merge happens has helped to ensure that mistakes are found and rectified, thereby helping to improve the quality of our code.

Chapter 7

Conclusion

This project researches the energy consumption of different language constructs in C#, with a focus on concurrency. This work is accomplished by taking energy measurements on a PC running Linux, to facilitate the use of Running Average Power Limit (RAPL) readings for measuring energy consumption.

As the focus is on concurrency constructs, we have created 150 microbenchmarks that look at concurrency, this includes benchmarks for different workloads and benchmarks created for analysis of differences between different concurrency constructs. The validity of results is determined by performing two-sample t-tests, Kolmogorov-Smirnov tests, and sanity checks using the CPU's expected energy usage and the runtime of the benchmarks. The results are then used to analyze the cause of differing behaviors.

Based on the findings from our microbenchmarks, we select eight macrobenchmarks from Rosetta Code [2]. These serve as the base for refactorings and comparison of language construct behavior in macrobenchmarks. These benchmarks have required changes to conform to the requirements of the framework [3] and as most of these are intended for single-threaded use and have low runtimes, only measuring a few milliseconds, multiple benchmarks are refactored to facilitate larger workloads. Each is refactored with a combination of three different code changes, which are among the lowest energy-consuming changes in a particular category of language constructs, according current and previous research in [3]. This results in a total of 68 macrobenchmarks.

We have found that in most cases energy improvements are compounded, however, in some cases an improvement constitutes an insignificant part of

the energy changes, which makes it negligible compared to the remainder of the benchmark. We find, from both micro- and macrobenchmarks, that elapsed time and energy efficiency of concurrent benchmarks are heavily influenced by workload size. We also have that a threshold workload must be reached to see improvements when changing a sequential program into a concurrent one. Additionally, we have found that elapsed time and energy consumption of concurrent benchmarks do not necessarily correlate, a faster program does not always mean a more energy-efficient program.

In summary, we have made the following contributions in this project:

1. A suite of C# microbenchmarks for *concurrency*, *casting*, *parameter mechanisms* and *lambda* expressions,
2. results of these microbenchmarks and the reason behind the results,
3. a suite of C# macrobenchmarks that test generalization and combinations of different language constructs, and
4. results of these macrobenchmarks and an analysis of the observed behavior.

The most vital findings throughout this project are that we have observed small workloads cause concurrency to be ineffective in regard to energy efficiency, however, the degree varies between the different approaches. Also, optimizing for speed, concurrency pays off with smaller workloads in contrast to optimizing for energy efficiency. The changes that occur in microbenchmarks remain the same in macrobenchmarks, however, given the larger energy consumption of these, changes with relatively low impact end up resulting in negligible changes to energy consumption. We have discovered that in most cases the combination of different code changes resulted in compounding the improvements from the individual code changes.

Chapter 8

Future Work

We now look at future work based on the research conducted in this project. This is both regarding how the research we have conducted can be extended and used to help ensure better energy efficiency in programming.

8.1 Benchmarks

One area where there is room for extending the conducted research is in the benchmarks themselves. This concerns both the microbenchmarks and the macrobenchmarks. We present four subjects for which the benchmark research can be extended.

Categories

In this project, we look at microbenchmarks and macrobenchmarks. Another option that can be utilized is applications. Using applications and refactoring them will allow gathering insight into what language constructs, both on their own, and in combinations have a significant impact on the energy consumption of applications. This extends the knowledge we have gathered regarding language constructs with negligible impact on energy consumption.

Combinations

Looking at more than three combinations in conjunction will give more results and uncover potential interactions between more optimizations. We can add additional language constructs, which are considered when refactoring benchmarks. This would give us better insight into the effect of more possible combinations, as there might be certain changes that have more or less impact on each other than what has been seen throughout this project. To accomplish this, one could design and implement a tool that is capable of automatically detecting potential refactorings and then refactoring them. An example of similar work is [17], which presents a tool that accomplishes this regarding refactorings.

Metrics

Looking at a broader spectrum of metrics in detail like runtime and DRAM energy usage might give even more insight into the potential gains and detriments of different language constructs. We already have some cases with concurrency where we observe that energy consumption is roughly the same, however, there are significant gains when it concerns the elapsed time of the benchmarks.

Amount of parallelism

All of the microbenchmarks on concurrency are fully parallelizable, however, this might not be the case in real applications. As such more information can be gathered from benchmarks that are 90%, 80%, 70%, etc parallelizable, i.e. has a synchronization phase, to see the effects of concurrency here. These could be benchmarks that look at k-means clustering, raytracing, the Santa Claus problem, or similar work as discussed in Chapter 6.

8.2 Ideas for future study

Other potential future work might look at not extending what is done in this project, but instead, using it as the basis for research in another area or the creation of some resource that helps with improving the energy consumption of applications.

Construct Improvement

An area of research worth looking into is how to improve some of the language constructs we have utilized. One construct that could be of particular interest is Language Integrated Query (LINQ), as it has proven to be the cause of substantially higher energy consumption in all benchmarks where we have removed it. This could involve looking into what technologies inside of LINQ are actually the cause of the high energy consumption, and by looking at these see if they can be changed or exchanged to facilitate better energy efficiency.

Concurrency thresholds

There could also be research regarding the threshold of work required of the different concurrent constructs. Our work only establishes what this workload is for a particular method. However, it would be of interest to establish a method of determining workload size and use it to determine what concurrency constructs to use if any. Regarding this, a look at which mechanisms in concurrency cause a speedup and improved/worsened energy efficiency could be of interest. Also, if there is a particular concurrency construct that scales better with the number of cores, or whether the different constructs are better depending on the core count.

Maintainability

A well-suited area of research, which could be used in the extension of our findings, is maintainability. We have a specific focus on energy consumption, however, this could potentially incur costs for the development and maintenance of applications.

During this project, we have seen that `LINQ` hurts energy efficiency. `LINQ` is a library created to ease the construction of queries [79]. As one of the languages construct changes with the largest impact on energy consumption,

it would make sense to look at how its removal impacts the maintainability of the code, and the same could be done for concurrency. It might be the case that the different approaches have different effects on maintainability, and if that is the case it makes sense to look into how the degree of impact on maintainability and energy consumption correlates.

Tools

To test the effects of language constructs on a larger sample, a tool for detecting and refactoring these language changes can be created, an example of this is seen in [17]. This tool could be given a large database of C# programs and output all permutations of code changes. The tool could also be used on a single codebase and apply all code changes to increase the energy efficiency of that program. Another idea is creating a tool that is capable of identifying the workload of different segments of code, which could then be used to determine if parallelization of this particular segment would improve energy efficiency or elapsed time.

Bibliography

[1] Aleksander Øster Nielsen, Kasper Jepsen, and Rasmus Smit Lindholt. *Benchmarks-P10*. URL: https://gitlab.com/rasser900/Benchmarks-P10.

[2] Rosetta Code. *Rosetta Code*. URL: http://rosettacode.org/wiki/Rosetta_Code (visited on 2022-03-25).

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Chao Jin, Bronis R de Supinski, David Abramson, Heidi Poxon, Luiz DeRose, Minh Ngoc Dinh, Mark Endrei, and Elizabeth R Jessup. "A survey on software methods to improve the energy efficiency of parallel computing". In: *The International Journal of High Performance Computing Applications* 31.6 (2017), pp. 517–549. DOI: 10.1177/1094342016665471. eprint: https://doi.org/10.1177/1094342016665471. URL: https://doi.org/10.1177/1094342016665471.

[12] Garrison Prinslow. *Overview of Performance Measurement and Analytical Modeling Techniques for Multi-core Processors*. URL: https://www.cs.wustl.edu/~jain/cse567-11/ftp/multcore/ (visited on 2022-02-28).

[13] Semyon Khokhriakov, Ravi Reddy Manumachu, and Alexey Lastovetsky. "Multicore processor computing is not energy proportional: An opportunity for bi-objective optimization for energy and performance". In: *Applied Energy* 268 (2020), p. 114957. ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2020.114957. URL: https://www.sciencedirect.com/science/article/pii/S0306261920304694.

[14] TIOBE. *TIOBE Index for January 2022*. 2022-02. URL: https://www.tiobe.com/tiobe-index/ (visited on 2022-02-02).

[15] 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.

[16] Gustavo Pinto, Fernando Castor, and Yu David Liu. "Understanding Energy Behaviors of Thread Management Constructs". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 345–360. ISBN: 9781450325851. DOI: 10.1145/2660193.2660235. URL: https://doi.org/10.1145/2660193.2660235.

[17] 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.

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

[19] LLC. Axosoft. *GitKraken Boards*. 2021. URL: https://www.gitkraken.com/boards (visited on 2022-02-01).

[20] Garrison Prinslow. *Overview of Performance Measurement and Analytical Modeling Techniques for Multi-core Processors*. URL: https://www.cs.wustl.edu/~jain/cse567-11/ftp/multcore/ (visited on 2022-03-09).

[21] Dong Hyuk Woo and Hsien-Hsin S. Lee. "Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era". In: *Computer* 41.12 (2008), pp. 24–31. DOI: 10.1109/MC.2008.494.

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

[23] Microsoft. *Task Parallel Library (TPL)*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl (visited on 2022-02-09).

[24] Microsoft. *Thread Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=net-6.0 (visited on 2022-02-18).

[25] Microsoft. *Parallel.For Method*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.for?view=net-6.0 (visited on 2022-04-01).

[26] Microsoft. *Parallel Loops*. URL: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff963552(v=pandp.10) (visited on 2022-04-01).

[27] Microsoft. *.NET Performance Tips*. URL: https://docs.microsoft.com/en-us/dotnet/framework/performance/performance-tips#boxing-and-unboxing (visited on 2022-02-06).

[28] Microsoft. *Implicit conversions*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions#implicit-conversions (visited on 2022-02-14).

[29] Microsoft. *Cast expression*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/type-testing-and-cast#cast-expression (visited on 2022-02-09).

[30] Microsoft. *Explicit conversions*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions#explicit-conversions (visited on 2022-02-14).

[31] Microsoft. *as operator*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/type-testing-and-cast#as-operator (visited on 2022-02-09).

[32] Microsoft. *Passing Parameters (C# Programming Guide)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/passing-parameters (visited on 2022-02-14).

[33] Microsoft. *in parameter modifier (C# Reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in-parameter-modifier (visited on 2022-02-14).

[34] Microsoft. *ref (C# Reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref (visited on 2022-02-14).

[35] Microsoft. *out parameter modifier (C# Reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier (visited on 2022-02-14).

[36] Microsoft. *Lambda expressions (C# Programming Guide)*. 2022. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions/ (visited on 2022-02-04).

[37] Microsoft. *Delegates (C# Programming Guide)*. 2022. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/ (visited on 2022-02-04).

[38] Noah Gibbs. *Microbenchmarks vs Macrobenchmarks (i.e. What's a Microbenchmark?)* 2019. URL: https://engineering.appfolio.com/appfolio-engineering/2019/1/7/microbenchmarks-vs-macrobenchmarks-ie-whats-a-microbenchmark (visited on 2021-02-07).

[39] Nicolas Poggi. "Microbenchmark". In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Y. Zomaya. Cham: Springer International Publishing, 2019, pp. 1143–1152. ISBN: 978-3-319-77525-8. DOI: 10.1007/978-3-319-77525-8_111. URL: https://doi.org/10.1007/978-3-319-77525-8_111.

[40] Peter Sestoft. "Microbenchmarks in Java and C#". In: *Lecture Notes, September* (2013).

[41] Bartosz and Yoh Deadfall. *PowerUp*. URL: https://github.com/badamczewski/PowerUp (visited on 2022-02-11).

[42] 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.

[43] 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 2022-02-07).

[44] Frank J Massey Jr. "The Kolmogorov-Smirnov test for goodness of fit". In: *Journal of the American statistical Association* 46.253 (1951), pp. 68–78.

[45] accord-net. *TwoSampleKolmogorovSmirnovTest Class*. URL: http://accord-framework.net/docs/html/T_Accord_Statistics_Testing_TwoSampleKolmogorovSmirnovT.htm (visited on 2022-03-29).

[46] Microsoft. *ParallelOptions.MaxDegreeOfParallelism Property*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.paralleloptions.maxdegreeofparallelism?view=net-6.0 (visited on 2022-02-24).

[47] Microsoft. *Conversions*. 2022. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/conversions (visited on 2022-02-23).

[48] Microsoft. *Action<T> Delegate*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.action-1?view=net-6.0 (visited on 2022-02-23).

[49] Microsoft. */System/Threading/Tasks/Parallel.cs*. URL: https://github.com/dotnet/runtime/blob/v6.0.1/src/libraries/System.Threading.Tasks.Parallel/src/System/Threading/Tasks/Parallel.cs (visited on 2022-03-28).

[50] Microsoft. *System/Threading/Tasks/ParallelRangeManager.cs*. URL: https://github.com/dotnet/runtime/blob/v6.0.1/src/libraries/System.Threading.Tasks.Parallel/src/System/Threading/Tasks/ParallelRangeManager.cs (visited on 2022-03-29).

[51] Microsoft. *System/Threading/Tasks/TaskReplicator.cs*. URL: https://github.com/dotnet/runtime/blob/v6.0.1/src/libraries/System.Threading.Tasks.Parallel/src/System/Threading/Tasks/TaskReplicator.cs (visited on 2022-03-31).

[52] Microsoft. *CountdownEvent Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.threading.countdownevent?view=net-6.0 (visited on 2022-05-03).

[53] 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 2022-03-02).

[54] 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 2022-03-02).

[55] Microsoft. *Explicit conversions*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/conversions#103-explicit-conversions (visited on 2022-03-03).

[56] Microsoft. *OpCodes.Isinst Field*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.isinst?view=net-6.0#remarks (visited on 2022-03-03).

[57] Microsoft. *OpCodes.Isinst Field*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.castclass?view=net-6.0#remarks (visited on 2022-03-03).

[58] Microsoft. *IsReadOnlyAttribute Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.isreadonlyattribute?view=net-6.0 (visited on 2022-03-14).

[59] Microsoft. *OpCodes.Ldloc Field*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.ldloc?view=net-6.0 (visited on 2022-04-05).

[60] Microsoft. *OpCodes.Stloc Field*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.stloc?view=net-6.0 (visited on 2022-04-05).

[61] Microsoft. *OpCodes.Ldloca_S Field*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.ldloca_s?view=net-6.0 (visited on 2022-04-05).

[62] Microsoft. *OpCodes.Ldarga_S Field*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.ldarga_s?view=net-6.0 (visited on 2022-05-03).

[63] Microsoft. *OpCodes.Ldind_I8 Field*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.ldarga_s?view=net-6.0 (visited on 2022-05-03).

[64] Microsoft. *OpCodes.Ret Field*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.ret?view=net-6.0 (visited on 2022-04-07).

[65] Microsoft. *Func<TResult> Delegate*. URL: https://docs.microsoft.com/en-us/dotnet/api/System.Func-1?view=net-6.0 (visited on 2022-03-08).

[66] 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).

[67] Microsoft. *ReadyToRun Compilation*. URL: https://docs.microsoft.com/en-us/dotnet/core/deploying/ready-to-run (visited on 2022-03-09).

[68] Rosetta Code. *100 Prisoners*. URL: https://rosettacode.org/wiki/100_prisoners#C.23 (visited on 2022-04-11).

[69] Aleksander Øster Nielsen, Kasper Jepsen, Lasse Stig Emil Rasmussen, Milton Kristian Lindof, Rasmus Smit Lindholt, Søren Bech Christensen, Lars Rechter, Martin Jensen, Casper Susgaard Nielsen, Anne Benedicte Abildgaard Ejsing, and Jacob Ruberg Nørhave. *CsharpRAPL*. URL: https://gitlab.com/Plagiatdrengene/CsharpRAPL/.

[70] Rosetta Code. *AlmostPrime*. URL: http://rosettacode.org/wiki/Almost_prime#C.23 (visited on 2022-05-02).

[71] Rosetta Code. *Chebyshev coefficients*. URL: http://rosettacode.org/wiki/Chebyshev_coefficients#C.23 (visited on 2022-05-02).

[72] Rosetta Code. *Compare length of two strings*. URL: http://rosettacode.org/wiki/Compare_length_of_two_strings#C.23 (visited on 2022-05-02).

[73] Rosetta Code. *Dijkstra algorithm*. URL: http://rosettacode.org/wiki/Dijkstra%5C%27s_algorithm#C.23 (visited on 2022-05-02).

[74] Rosetta Code. *Four rings or Four squares puzzle*. URL: http://rosettacode.org/wiki/4-rings_or_4-squares_puzzle#C.23 (visited on 2022-05-02).

[75] Rosetta Code. *Numbrix Puzzle*. URL: http://rosettacode.org/wiki/Solve_a_Numbrix_puzzle#C.23 (visited on 2022-05-02).

[76] Rosetta Code. *Sum to 100*. URL: http://rosettacode.org/wiki/Sum_to_100#C.23 (visited on 2022-05-02).

[77] Microsoft. *Array.IndexOf Method*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.array.indexof?view=net-6.0 (visited on 2022-05-10).

[78] Microsoft. *var pattern*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns#var-pattern (visited on 2022-05-09).

[79] Microsoft. *LINQ*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/ (visited on 2022-05-20).

[80] 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 2022-02-01).

Appendix A

Appendix

A.1 Framework Sanity Checks

The work presented here is used in Section 3.3.

Sanity check are performed by utilizing the elapsed time of benchmarks and power draw of the Intel Xeon W-1250P processor used in [3]. The Intel Xeon W-1250P has a Thermal Design Power (TDP) of 125 Watts, however given that the clock rate is effectively static and boosting has been turned off [3, p. 41], the expected TDP is 95 Watts [80]. As the Intel Xeon W-1250P has six cores the expected power draw of a single core is: $\frac{95 \text{ Watts}}{6} = 15,83$ Watts. This is utilized with the elapsed time of benchmarks to get a rough estimate of consumed energy. The formula used is Equation 3.1 in Section 3.3.

We look at one benchmark result from each group, or subgroup if one such exists, presented in [3, p. 60-18]. In Table A.1 the results of these sanity checks are seen. The difference between the rough estimate and observed result has been calculated in percent as well as the average difference.

Benchmark	Elapsed Time (ms)	Package Energy (μJ)	Rough Estimate (μJ)	Percentage Difference
Primitive Integer: Uint	2,935853	25.862,204	46.474,553	56,99%
Selection Switch: Switch	1,618432	18.620,357	25.619,779	31,64%
Selection Const: Switch Const Number	0,909927	10.226,356	14.404,144	33,92%
Selection Conditional: Conditional Operator	0,897267	9.355,048	14.203,737	41,16%
Selection If Statements: If	1,096640	11.151,124	17.359,811	43,55%
Collections List Get: Array Get	617,467346	8.794.373,745	9.774.508,090	10,56%
Collections List Removal: Immutable List Removal	728.103,041858	10.353.928.913,417	11.525.871.200	10,71%
Collections Table Get: Dictionary Get	4.843,913710	69.332.938,639	76.679.154	10,06%
String Concatenation: String Builder	42,375883	649.596,221	670.810,228	3,21%
Invocation Local Function: Local Function Invocation	1,705528	17.748,136	26.998,508	41,35%
Invocation Reflection: Reflection Delegate	2,202594	29.323,287	34.867,063	17,27%
Objects: Struct Method	0,978953	9.201,011	15.496,826	50,98%
Inheritance: Inheritance Virtual	1,223562	17.313,036	19.368,987	11,21%
Exceptions `Try catch` with exceptions: Try Catch All E	2.005,725403	28.131.047,363	31.750.633,100	12,09%
Exceptions `Try catch` without exceptions: Try Finally Equiv No E	3,546218	31.294,703	56.136,631	56,83%
Average	-	-	-	28,77%

Table A.1: Table showing old results and expected results according to Equation 3.1, along with the percentage difference between results recorded in [3] and the rough estimate.

A.2 *P*-Values of Microbenchmarks

Here is a presentation of the *p*-values from from the microbenchmark results, it is used in Section 4.1.

A.2.1 Concurrency

Elapsed Time *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 1	Sequential	Thread Pool 1
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 1	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.2: Table showing the *p*-values for the group Concurrency 1 with regards to Elapsed Time.

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 1	Sequential	Thread Pool 1
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 1	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.3: Table showing the *p*-values for the group Concurrency 1 with regards to Package Energy.

DRAM Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

DRAM Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 1	Sequential	Thread Pool 1
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 1	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.4: Table showing the *p*-values for the group Concurrency 1 with regards to DRAM Energy.

Elapsed Time *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,104
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 10	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 10	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 10	Sequential	Thread Pool 10
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	0,104	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 10	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 10	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.5: Table showing the *p*-values for the group Concurrency 10 with regards to Elapsed Time.

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,263
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 10	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 10	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 10	Sequential	Thread Pool 10
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	0,263	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 10	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 10	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.6: Table showing the *p*-values for the group Concurrency 10 with regards to Package Energy.

DRAM Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,103
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 10	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 10	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

DRAM Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 10	Sequential	Thread Pool 10
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	0,103	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 10	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 10	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.7: Table showing the *p*-values for the group Concurrency 10 with regards to DRAM Energy.

Elapsed Time *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,524	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	0,524	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 100	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 100	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 100	Sequential	Thread Pool 100
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 100	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 100	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.8: Table showing the *p*-values for the group Concurrency 100 with regards to Elapsed Time.

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	0,750	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 100	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 100	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 100	Sequential	Thread Pool 100
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	0,750	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 100	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 100	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.9: Table showing the *p*-values for the group Concurrency 100 with regards to Package Energy.

DRAM Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,101	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	0,101	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 100	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 100	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

DRAM Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 100	Sequential	Thread Pool 100
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 100	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 100	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.10: Table showing the *p*-values for the group Concurrency 100 with regards to DRAM Energy.

Elapsed Time *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,076	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	0,567	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 1000	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 1000	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 1000	Sequential	Thread Pool 1000
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	0,567	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	0,076	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	0,113
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 1000	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 1000	<0,05	<0,05	<0,05	0,113	<0,05	<0,05	<0,05	-

Table A.11: Table showing the *p*-values for the group Concurrency 1000 with regards to Elapsed Time.

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,137	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 1000	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 1000	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 1000	Sequential	Thread Pool 1000
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	0,137	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 1000	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 1000	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.12: Table showing the *p*-values for the group Concurrency 1000 with regards to Package Energy.

DRAM Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	0,688	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 1000	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 1000	<0,05	0,397	0,091	<0,05	<0,05	<0,05	<0,05	<0,05

DRAM Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 1000	Sequential	Thread Pool 1000
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,397
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,091
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	0,688	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 1000	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 1000	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.13: Table showing the *p*-values for the group Concurrency 1000 with regards to DRAM Energy.

Elapsed Time *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,295	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,183
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,156	<0,05
Parallel for 6t	0,295	<0,05	<0,05	<0,05	<0,05	0,156	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	0,183	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,098
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	0,392	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 10k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 10k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 10k	Sequential	Thread Pool 10k
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	0,392	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	0,098	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 10k	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 10k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.14: Table showing the *p*-values for the group Concurrency 10k with regards to Elapsed Time.

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	0,647	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 10k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	0,738	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 10k	<0,05	<0,05	0,566	<0,05	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 10k	Sequential	Thread Pool 10k
Parallel for Default	0,647	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,738	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,566
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 10k	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 10k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.15: Table showing the *p*-values for the group Concurrency 10k with regards to Package Energy.

DRAM Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,386	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,197
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,156	<0,05
Parallel for 6t	0,386	<0,05	<0,05	<0,05	<0,05	0,156	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	0,197	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,094
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	0,318	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 10k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 10k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

DRAM Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 10k	Sequential	Thread Pool 10k
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	0,318	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	0,094	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 10k	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 10k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.16: Table showing the *p*-values for the group Concurrency 10k with regards to DRAM Energy.

Elapsed Time *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	0,910	0,661	0,075	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	0,910	<0,05	<0,05	<0,05	-	0,051	0,729	<0,05
Manual Thread 4	0,661	<0,05	<0,05	<0,05	0,051	-	0,972	<0,05
Parallel for 6t	0,075	<0,05	<0,05	<0,05	0,729	0,972	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,939
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 100k	<0,05	<0,05	<0,05	<0,05	0,586	0,787	0,242	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 100k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 100k	Sequential	Thread Pool 100k
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	0,586	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	0,787	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	0,242	<0,05	<0,05
Manual Thread 6	0,939	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,551
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 100k	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 100k	<0,05	0,551	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.17: Table showing the *p*-values for the group Concurrency 100k with regards to Elapsed Time.

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	0,061	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,593	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	0,061	0,593	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	0,228	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 100k	<0,05	<0,05	<0,05	<0,05	<0,05	0,229	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 100k	<0,05	<0,05	<0,05	0,059	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 100k	Sequential	Thread Pool 100k
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	0,228	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,059
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	0,229	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 100k	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 100k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.18: Table showing the *p*-values for the group Concurrency 100k with regards to Package Energy.

DRAM Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	0,935	0,641	0,070	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	0,935	<0,05	<0,05	<0,05	-	<0,05	0,701	<0,05
Manual Thread 4	0,641	<0,05	<0,05	<0,05	<0,05	-	0,986	<0,05
Parallel for 6t	0,070	<0,05	<0,05	<0,05	0,701	0,986	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,943
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 100k	<0,05	<0,05	<0,05	<0,05	0,210	0,300	<0,05	0,637
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 100k	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

DRAM Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 100k	Sequential	Thread Pool 100k
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	0,210	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	0,300	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	0,943	<0,05	<0,05	<0,05	<0,05	0,637	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	0,294	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,740
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 100k	0,294	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 100k	<0,05	0,740	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.19: Table showing the *p*-values for the group Concurrency 100k with regards to DRAM Energy.

Elapsed Time *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	0,299	0,272	0,464	<0,05
Parallel for 2t	<0,05	-	0,677	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	0,677	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	0,299	<0,05	<0,05	<0,05	-	0,555	0,137	<0,05
Manual Thread 4	0,272	<0,05	<0,05	<0,05	0,555	-	<0,05	<0,05
Parallel for 6t	0,464	<0,05	<0,05	<0,05	0,137	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	0,233	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,172
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 1mil	0,228	<0,05	<0,05	<0,05	0,493	0,448	0,701	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 1mil	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 1mil	Sequential	Thread Pool 1mil
Parallel for Default	0,233	<0,05	<0,05	<0,05	<0,05	0,228	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	0,493	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	0,448	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	0,701	<0,05	<0,05
Manual Thread 6	0,172	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	0,117	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 1mil	0,117	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 1mil	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.20: Table showing the *p*-values for the group Concurrency 1mil with regards to Elapsed Time.

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 96	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	0,103	0,945	0,343	<0,05
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	0,332	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	0,103	<0,05	<0,05	<0,05	-	0,074	<0,05	<0,05
Manual Thread 4	0,945	<0,05	<0,05	<0,05	0,074	-	0,499	<0,05
Parallel for 6t	0,343	<0,05	<0,05	<0,05	<0,05	0,499	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 1mil	<0,05	<0,05	<0,05	<0,05	<0,05	0,421	0,971	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 1mil	<0,05	0,289	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 2	Manual Thread 192	Using Tasks 1mil	Sequential	Thread Pool 1mil
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	0,332	<0,05	<0,05	<0,05	0,289
Manual Thread 2	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	0,367
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	0,421	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	0,971	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,120
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 1mil	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 1mil	<0,05	0,120	<0,05	0,367	<0,05	<0,05	<0,05	-

Table A.21: Table showing the *p*-values for the group Concurrency 1mil with regards to Package Energy.

DRAM Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	0,304	0,293	0,466	<0,05
Parallel for 2t	<0,05	-	1,000	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	1,000	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	0,304	<0,05	<0,05	<0,05	-	0,664	0,123	<0,05
Manual Thread 4	0,293	<0,05	<0,05	<0,05	0,664	-	<0,05	<0,05
Parallel for 6t	0,466	<0,05	<0,05	<0,05	0,123	<0,05	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	0,238	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,195
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 1mil	0,248	<0,05	<0,05	<0,05	0,486	0,463	0,690	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 1mil	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

DRAM Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 1mil	Sequential	Thread Pool 1mil
Parallel for Default	0,238	<0,05	<0,05	<0,05	<0,05	0,248	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	0,486	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	0,463	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	0,690	<0,05	<0,05
Manual Thread 6	0,195	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	0,121	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Using Tasks 1mil	0,121	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 1mil	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.22: Table showing the *p*-values for the group Concurrency 1mil with regards to DRAM Energy.

Elapsed Time *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	0,108	<0,05	0,443
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	0,397	<0,05	<0,05
Manual Thread 4	0,108	<0,05	<0,05	<0,05	0,397	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,258
Manual Thread 6	0,443	<0,05	<0,05	<0,05	<0,05	<0,05	0,258	-
Manual Thread 12	0,068	<0,05	<0,05	<0,05	<0,05	<0,05	0,530	0,063
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 10mil	<0,05	<0,05	<0,05	<0,05	0,420	0,492	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 10mil	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 10mil	Sequential	Thread Pool 10mil
Parallel for Default	0,068	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	0,420	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	0,492	<0,05	<0,05
Parallel for 6t	0,530	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	0,063	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	0,288	0,997	<0,05	0,099	<0,05
Manual Thread 96	<0,05	<0,05	0,288	-	0,337	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	0,997	0,337	-	<0,05	0,114	<0,05
Using Tasks 10mil	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	0,099	<0,05	0,114	<0,05	-	<0,05
Thread Pool 10mil	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.23: Table showing the *p*-values for the group Concurrency 10mil with regards to Elapsed Time.

Package Energy *p*-Values	Manual Thread 24	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	<0,05	<0,05	<0,05	<0,05	<0,05	0,055	<0,05	<0,05
Parallel for 2t	<0,05	-	0,053	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	0,053	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,705	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	0,705	-	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-
Manual Thread 12	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 10mil	<0,05	<0,05	<0,05	<0,05	<0,05	0,800	0,560	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 10mil	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Manual Thread 12	Parallel for Default	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 10mil	Sequential	Thread Pool 10mil
Parallel for Default	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 4	<0,05	0,055	<0,05	<0,05	<0,05	0,800	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	0,560	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	<0,05	<0,05	<0,05	0,919	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	-	0,091	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	0,091	-	<0,05	<0,05	<0,05
Using Tasks 10mil	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	0,919	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 10mil	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.24: Table showing the *p*-values for the group Concurrency 10mil with regards to Package Energy.

DRAM Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	<0,05	<0,05	<0,05	<0,05	0,116	<0,05	0,465
Parallel for 2t	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	-	0,398	<0,05	<0,05
Manual Thread 4	0,116	<0,05	<0,05	<0,05	0,398	-	<0,05	<0,05
Parallel for 6t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,265
Manual Thread 6	0,465	<0,05	<0,05	<0,05	<0,05	<0,05	0,265	-
Manual Thread 12	0,070	<0,05	<0,05	<0,05	<0,05	<0,05	0,599	<0,05
Manual Thread 24	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Using Tasks 10mil	<0,05	<0,05	<0,05	<0,05	0,412	0,492	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Thread Pool 10mil	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

DRAM Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks 10mil	Sequential	Thread Pool 10mil
Parallel for Default	0,070	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 3t	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 4t	<0,05	<0,05	<0,05	<0,05	<0,05	0,412	<0,05	<0,05
Manual Thread 4	<0,05	<0,05	<0,05	<0,05	<0,05	0,492	<0,05	<0,05
Parallel for 6t	0,599	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 6	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 12	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 24	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread 48	<0,05	<0,05	-	0,261	0,066	<0,05	<0,05	<0,05
Manual Thread 96	<0,05	<0,05	0,261	-	0,707	<0,05	<0,05	<0,05
Manual Thread 192	<0,05	<0,05	0,066	0,707	-	<0,05	<0,05	<0,05
Using Tasks 10mil	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sequential	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Thread Pool 10mil	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.25: Table showing the *p*-values for the group Concurrency 10mil with regards to DRAM Energy.

A.2.2 Numeric Casting

Elapsed Time *p*-Values	Implicit	Explicit
Implicit	-	0,808
Explicit	0,808	-

Table A.26: Table showing the *p*-values for the group Casting with regards to Elapsed Time.

Package Energy *p*-Values	Implicit	Explicit
Implicit	-	<0,05
Explicit	<0,05	-

Table A.27: Table showing the *p*-values for the group Casting with regards to Package Energy.

DRAM Energy *p*-Values	Implicit	Explicit
Implicit	-	0,863
Explicit	0,863	-

Table A.28: Table showing the *p*-values for the group Casting with regards to DRAM Energy.

A.2.3 Derived Casting

Elapsed Time *p*-Values	Reference Explicit	As
Reference Explicit	-	0,526
As	0,526	-

Table A.29: Table showing the *p*-values for the group Reference Casting with regards to Elapsed Time.

Package Energy *p*-Values	Reference Explicit	As
Reference Explicit	-	<0,05
As	<0,05	-

Table A.30: Table showing the *p*-values for the group Reference Casting with regards to Package Energy.

DRAM Energy *p*-Values	Reference Explicit	As
Reference Explicit	-	0,334
As	0,334	-

Table A.31: Table showing the *p*-values for the group Reference Casting with regards to DRAM Energy.

A.2.4 Non-Modifying Parameter Mechanisms

Elapsed Time *p*-Values	In	Ref
In	-	0,883
Ref	0,883	-

Table A.32: Table showing the *p*-values for the group Param Mech with regards to Elapsed Time.

Package Energy *p*-Values	In	Ref
In	-	<0,05
Ref	<0,05	-

Table A.33: Table showing the *p*-values for the group Param Mech with regards to Package Energy.

DRAM Energy *p*-Values	In	Ref
In	-	0,693
Ref	0,693	-

Table A.34: Table showing the *p*-values for the group Param Mech with regards to DRAM Energy.

A.2.5 Modifying Parameter Mechanisms

Elapsed Time *p*-Values	Ref Mod	Out
Ref Mod	-	0,105
Out	0,105	-

Table A.35: Table showing the *p*-values for the group Param Mech Mod with regards to Elapsed Time.

Package Energy *p*-Values	Ref Mod	Out
Ref Mod	-	<0,05
Out	<0,05	-

Table A.36: Table showing the *p*-values for the group Param Mech Mod with regards to Package Energy.

DRAM Energy *p*-Values	Ref Mod	Out
Ref Mod	-	0,196
Out	0,196	-

Table A.37: Table showing the *p*-values for the group Param Mech Mod with regards to DRAM Energy.

A.2.6 Modifying value-type

Elapsed Time *p*-Values	Return Large Struct	Ref Large Struct	Return Small Struct	Ref Small Struct
Return Large Struct	-	<0,05	<0,05	<0,05
Ref Large Struct	<0,05	-	<0,05	<0,05
Return Small Struct	<0,05	<0,05	-	<0,05
Ref Small Struct	<0,05	<0,05	<0,05	-

Table A.38: Table showing the *p*-values for the group Param Mech Return Struct with regards to Elapsed Time.

Package Energy *p*-Values	Return Large Struct	Ref Large Struct	Return Small Struct	Ref Small Struct
Return Large Struct	-	<0,05	<0,05	<0,05
Ref Large Struct	<0,05	-	<0,05	<0,05
Return Small Struct	<0,05	<0,05	-	<0,05
Ref Small Struct	<0,05	<0,05	<0,05	-

Table A.39: Table showing the *p*-values for the group Param Mech Return Struct with regards to Package Energy.

DRAM Energy *p*-Values	Return Large Struct	Ref Large Struct	Return Small Struct	Ref Small Struct
Return Large Struct	-	<0,05	<0,05	<0,05
Ref Large Struct	<0,05	-	<0,05	<0,05
Return Small Struct	<0,05	<0,05	-	<0,05
Ref Small Struct	<0,05	<0,05	<0,05	-

Table A.40: Table showing the *p*-values for the group Param Mech Return Struct with regards to DRAM Energy.

A.2.7 Returning Parameter Mechanisms

Elapsed Time *p*-Values	Ref Return	Out Return	Return	Return Alt
Ref Return	-	0,536	0,121	0,342
Out Return	0,536	-	0,195	0,694
Return	0,121	0,195	-	0,309
Return Alt	0,342	0,694	0,309	-

Table A.41: Table showing the *p*-values for the group Param Mech Return with regards to Elapsed Time.

Package Energy *p*-Values	Ref Return	Out Return	Return	Return Alt
Ref Return	-	<0,05	<0,05	<0,05
Out Return	<0,05	-	<0,05	<0,05
Return	<0,05	<0,05	-	<0,05
Return Alt	<0,05	<0,05	<0,05	-

Table A.42: Table showing the *p*-values for the group Param Mech Return with regards to Package Energy.

DRAM Energy *p*-Values	Ref Return	Out Return	Return	Return Alt
Ref Return	-	0,885	0,855	0,878
Out Return	0,885	-	0,634	0,969
Return	0,855	0,634	-	0,646
Return Alt	0,878	0,969	0,646	-

Table A.43: Table showing the *p*-values for the group Param Mech Return with regards to DRAM Energy.

A.2.8 Lambda

Elapsed Time *p*-Values	Lambda	Lambda Closure	Lambda Param	Lambda Delegate	Lambda Delegate Closure	Lambda Action
Lambda	-	<0,05	0,448	<0,05	<0,05	0,237
Lambda Closure	<0,05	-	<0,05	0,270	<0,05	<0,05
Lambda Param	0,448	<0,05	-	<0,05	<0,05	0,097
Lambda Delegate	<0,05	0,270	<0,05	-	<0,05	<0,05
Lambda Delegate Closure	<0,05	<0,05	<0,05	<0,05	-	<0,05
Lambda Action	0,237	<0,05	0,097	<0,05	<0,05	-

Table A.44: Table showing the *p*-values for the group Lambda Expression with regards to Elapsed Time.

Package Energy *p*-Values	Lambda	Lambda Closure	Lambda Param	Lambda Delegate	Lambda Delegate Closure	Lambda Action
Lambda	-	<0,05	<0,05	<0,05	<0,05	<0,05
Lambda Closure	<0,05	-	<0,05	<0,05	<0,05	<0,05
Lambda Param	<0,05	<0,05	-	<0,05	<0,05	<0,05
Lambda Delegate	<0,05	<0,05	<0,05	-	<0,05	<0,05
Lambda Delegate Closure	<0,05	<0,05	<0,05	<0,05	-	<0,05
Lambda Action	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.45: Table showing the *p*-values for the group Lambda Expression with regards to Package Energy.

DRAM Energy *p*-Values	Lambda	Lambda Closure	Lambda Param	Lambda Delegate	Lambda Delegate Closure	Lambda Action
Lambda	-	<0,05	0,351	<0,05	<0,05	0,054
Lambda Closure	<0,05	-	<0,05	0,138	<0,05	<0,05
Lambda Param	0,351	<0,05	-	<0,05	<0,05	0,655
Lambda Delegate	<0,05	0,138	<0,05	-	<0,05	<0,05
Lambda Delegate Closure	<0,05	<0,05	<0,05	<0,05	-	<0,05
Lambda Action	0,054	<0,05	0,655	<0,05	<0,05	-

Table A.46: Table showing the *p*-values for the group Lambda Expression with regards to DRAM Energy.

A.3 *P*-Values of Macrobenchmarks

Here is a presentation of the *p*-values from from the macrobenchmark results, it is used in Section 5.2.

A.3.1 100 Prisoners

Elapsed Time *p*-Values	Original	Parallel for	No LINQ	Array	Array No LINQ	Parallel for No LINQ	Parallel for Array	Parallel for Array No LINQ
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
No LINQ	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Array	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Array No LINQ	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for No LINQ	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for Array	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Parallel for Array No LINQ	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.47: Table showing the *p*-values for the group Prisoners with regards to Elapsed Time.

Package Energy *p*-Values	Original	Parallel for	No LINQ	Array	Array No LINQ	Parallel for No LINQ	Parallel for Array	Parallel for Array No LINQ
Original	-	<0,05	<0,05	0,169	<0,05	<0,05	<0,05	<0,05
Parallel for	<0,05	-	<0,05	<0,05	<0,05	<0,05	0,318	<0,05
No LINQ	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Array	0,169	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Array No LINQ	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for No LINQ	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for Array	<0,05	0,318	<0,05	<0,05	<0,05	<0,05	-	<0,05
Parallel for Array No LINQ	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.48: Table showing the *p*-values for the group Prisoners with regards to Package Energy.

DRAM Energy *p*-Values	Original	Parallel for	No LINQ	Array	Array No LINQ	Parallel for No LINQ	Parallel for Array	Parallel for Array No LINQ
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
No LINQ	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Array	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Array No LINQ	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for No LINQ	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for Array	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Parallel for Array No LINQ	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.49: Table showing the *p*-values for the group Prisoners with regards to DRAM Energy.

A.3.2 Almost Prime

Elapsed Time *p*-Values	Original	Manual Thread	Array	Struct	Manual Thread Array	Manual Thread Struct	Array Struct	Manual Thread Array Struct
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread	<0,05	-	<0,05	<0,05	0,934	0,244	<0,05	0,713
Array	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Struct	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread Array	<0,05	0,934	<0,05	<0,05	-	0,219	<0,05	0,781
Manual Thread Struct	<0,05	0,244	<0,05	<0,05	0,219	-	<0,05	0,132
Array Struct	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread Array Struct	<0,05	0,713	<0,05	<0,05	0,781	0,132	<0,05	-

Table A.50: Table showing the *p*-values for the group Almost Prime with regards to Elapsed Time.

Package Energy *p*-Values	Original	Manual Thread	Array	Struct	Manual Thread Array	Manual Thread Struct	Array Struct	Manual Thread Array Struct
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,648
Array	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Struct	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Manual Thread Array	<0,05	<0,05	<0,05	<0,05	-	0,487	<0,05	<0,05
Manual Thread Struct	<0,05	<0,05	<0,05	<0,05	0,487	-	<0,05	<0,05
Array Struct	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread Array Struct	<0,05	0,648	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.51: Table showing the *p*-values for the group Almost Prime with regards to Package Energy.

DRAM Energy *p*-Values	Original	Manual Thread	Array	Struct	Manual Thread Array	Manual Thread Struct	Array Struct	Manual Thread Array Struct
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Manual Thread	<0,05	-	<0,05	<0,05	0,870	0,144	<0,05	0,906
Array	<0,05	<0,05	-	0,281	<0,05	<0,05	<0,05	<0,05
Struct	<0,05	<0,05	0,281	-	<0,05	<0,05	<0,05	<0,05
Manual Thread Array	<0,05	0,870	<0,05	<0,05	-	0,212	<0,05	0,797
Manual Thread Struct	<0,05	0,144	<0,05	<0,05	0,212	-	<0,05	0,143
Array Struct	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Manual Thread Array Struct	<0,05	0,906	<0,05	<0,05	0,797	0,143	<0,05	-

Table A.52: Table showing the *p*-values for the group Almost Prime with regards to DRAM Energy.

A.3.3 Chebyshev coefficients

Elapsed Time *p*-Values	Original	Concurrent	Array	Datatype	Concurrent Array	Concurrent Datatype	Array Datatype	Concurrent Array Datatype
Original	-	<0,05	<0,05	0,371	<0,05	<0,05	0,288	<0,05
Concurrent	<0,05	-	<0,05	<0,05	0,108	<0,05	<0,05	0,069
Array	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Datatype	0,371	<0,05	<0,05	-	<0,05	<0,05	0,404	<0,05
Concurrent Array	<0,05	0,108	<0,05	<0,05	-	<0,05	<0,05	<0,05
Concurrent Datatype	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	0,484
Array Datatype	0,288	<0,05	<0,05	0,404	<0,05	<0,05	-	<0,05
Concurrent Array Datatype	<0,05	0,069	<0,05	<0,05	<0,05	0,484	<0,05	-

Table A.53: Table showing the *p*-values for the group Chebyshev with regards to Elapsed Time.

Package Energy *p*-Values	Original	Concurrent	Array	Datatype	Concurrent Array	Concurrent Datatype	Array Datatype	Concurrent Array Datatype
Original	-	<0,05	<0,05	0,300	<0,05	<0,05	<0,05	<0,05
Concurrent	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Array	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Datatype	0,300	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Concurrent Array	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Concurrent Datatype	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Array Datatype	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Concurrent Array Datatype	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.54: Table showing the *p*-values for the group Chebyshev with regards to Package Energy.

DRAM Energy *p*-Values	Original	Concurrent	Array	Datatype	Concurrent Array	Concurrent Datatype	Array Datatype	Concurrent Array Datatype
Original	-	<0,05	<0,05	0,270	<0,05	<0,05	0,098	<0,05
Concurrent	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,090
Array	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Datatype	0,270	<0,05	<0,05	-	<0,05	<0,05	0,183	<0,05
Concurrent Array	<0,05	<0,05	<0,05	<0,05	-	0,297	<0,05	0,448
Concurrent Datatype	<0,05	<0,05	<0,05	<0,05	0,297	-	<0,05	<0,05
Array Datatype	0,098	<0,05	<0,05	0,183	<0,05	<0,05	-	<0,05
Concurrent Array Datatype	<0,05	0,090	<0,05	<0,05	0,448	<0,05	<0,05	-

Table A.55: Table showing the *p*-values for the group Chebyshev with regards to DRAM Energy.

A.3.4 Compare length of two strings

Elapsed Time *p*-Values	Original	Selection	Parallel for 1	Parallel for 2	Foreach	Selection Foreach
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05
Selection	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 1	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for 2	<0,05	<0,05	<0,05	-	<0,05	<0,05
Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05
Selection Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	-
Selection Parallel for 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 1 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 1 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 2 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	Selection Parallel for 1	Selection Parallel for 2	Parallel for 1 Foreach	Parallel for 2 Foreach	Selection Parallel for 1 Foreach	Selection Parallel for 2 Foreach
Original	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 1	-	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 2	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 1 Foreach	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for 2 Foreach	<0,05	<0,05	<0,05	-	<0,05	0,160
Selection Parallel for 1 Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05
Selection Parallel for 2 Foreach	<0,05	<0,05	<0,05	0,160	<0,05	-

Table A.56: Table showing the *p*-values for the group Compare String Length with regards to Elapsed Time.

Package Energy *p*-Values	Original	Selection	Parallel for 1	Parallel for 2	Foreach	Selection Foreach
Original	-	0,449	<0,05	<0,05	<0,05	<0,05
Selection	0,449	-	<0,05	<0,05	<0,05	<0,05
Parallel for 1	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for 2	<0,05	<0,05	<0,05	-	<0,05	<0,05
Foreach	<0,05	<0,05	<0,05	<0,05	-	0,891
Selection Foreach	<0,05	<0,05	<0,05	<0,05	0,891	-
Selection Parallel for 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 1 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 1 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 2 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Selection Parallel for 1	Selection Parallel for 2	Parallel for 1 Foreach	Parallel for 2 Foreach	Selection Parallel for 1 Foreach	Selection Parallel for 2 Foreach
Original	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 1	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 1	-	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 2	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 1 Foreach	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for 2 Foreach	<0,05	<0,05	<0,05	-	<0,05	0,118
Selection Parallel for 1 Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05
Selection Parallel for 2 Foreach	<0,05	<0,05	<0,05	0,118	<0,05	-

Table A.57: Table showing the *p*-values for the group Compare String Length with regards to Package Energy.

DRAM Energy *p*-Values	Original	Selection	Parallel for 1	Parallel for 2	Foreach	Selection Foreach
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05
Selection	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for 1	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for 2	<0,05	<0,05	<0,05	-	<0,05	<0,05
Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05
Selection Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	-
Selection Parallel for 1	<0,05	<0,05	0,733	<0,05	<0,05	<0,05
Selection Parallel for 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 1 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 1 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 2 Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05

DRAM Energy *p*-Values	Selection Parallel for 1	Selection Parallel for 2	Parallel for 1 Foreach	Parallel for 2 Foreach	Selection Parallel for 1 Foreach	Selection Parallel for 2 Foreach
Original	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 1	0,733	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for 2	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 1	-	<0,05	<0,05	<0,05	<0,05	<0,05
Selection Parallel for 2	<0,05	-	<0,05	0,109	<0,05	0,154
Parallel for 1 Foreach	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for 2 Foreach	<0,05	0,109	<0,05	-	<0,05	0,408
Selection Parallel for 1 Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05
Selection Parallel for 2 Foreach	<0,05	0,154	<0,05	0,408	<0,05	-

Table A.58: Table showing the *p*-values for the group Compare String Length with regards to DRAM Energy.

A.3.5 Dijkstra

Elapsed Time *p*-Values	Original	No Linq	Array	Objects	No Linq Array	No Linq Objects	Array Objects	No Linq Array Objects
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
No Linq	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Array	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Objects	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
No Linq Array	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
No Linq Objects	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Array Objects	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
No Linq Array Objects	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.59: Table showing the *p*-values for the group Dijkstra with regards to Elapsed Time.

Package Energy *p*-Values	Original	No Linq	Array	Objects	No Linq Array	No Linq Objects	Array Objects	No Linq Array Objects
Original	-	<0,05	0,629	<0,05	<0,05	<0,05	<0,05	<0,05
No Linq	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	0,093
Array	0,629	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Objects	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
No Linq Array	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,449
No Linq Objects	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Array Objects	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
No Linq Array Objects	<0,05	0,093	<0,05	<0,05	0,449	<0,05	<0,05	-

Table A.60: Table showing the *p*-values for the group Dijkstra with regards to Package Energy.

DRAM Energy *p*-Values	Original	No Linq	Array	Objects	No Linq Array	No Linq Objects	Array Objects	No Linq Array Objects
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
No Linq	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Array	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Objects	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
No Linq Array	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
No Linq Objects	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Array Objects	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
No Linq Array Objects	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.61: Table showing the *p*-values for the group Dijkstra with regards to DRAM Energy.

A.3.6 Four Squares Puzzle

Elapsed Time *p*-Values	Original	Parallel for	No LINQ	Selection	Parallel for No LINQ	Parallel for Selection	No LINQ Selection	Parallel for No LINQ Selection
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for	<0,05	-	<0,05	<0,05	<0,05	0,554	<0,05	<0,05
No LINQ	<0,05	<0,05	-	<0,05	<0,05	<0,05	0,222	<0,05
Selection	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
for No LINQ	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,274
for Selection	<0,05	0,554	<0,05	<0,05	<0,05	-	<0,05	<0,05
No LINQ Selection	<0,05	<0,05	0,222	<0,05	<0,05	<0,05	-	<0,05
Parallel for No LINQ Selection	<0,05	<0,05	<0,05	<0,05	0,274	<0,05	<0,05	-

Table A.62: Table showing the *p*-values for the group Four Square Puzzle with regards to Elapsed Time.

Package Energy *p*-Values	Original	Parallel for	No LINQ	Selection	Parallel for No LINQ	Parallel for Selection	No LINQ Selection	Parallel for No LINQ Selection
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
No LINQ	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Selection	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Parallel for No LINQ	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for Selection	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
No LINQ Selection	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Parallel for No LINQ Selection	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.63: Table showing the *p*-values for the group Four Square Puzzle with regards to Package Energy.

DRAM Energy *p*-Values	Original	Parallel for	No LINQ	Selection	Parallel for No LINQ	Parallel for Selection	No LINQ Selection	Parallel for No LINQ Selection
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
No LINQ	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Selection	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
for No LINQ	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Parallel for Selection	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
No LINQ Selection	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Parallel for No LINQ Selection	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.64: Table showing the *p*-values for the group Four Square Puzzle with regards to DRAM Energy.

A.3.7 Numbrix Puzzle

Elapsed Time *p*-Values	Original	Linq	Parallel for	Selection	Linq Parallel for	Linq Selection	Parallel for Selection	Linq Parallel for Selection
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for	<0,05	<0,05	-	<0,05	<0,05	<0,05	0,117	<0,05
Selection	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Linq Parallel for	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,518
Linq Selection	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for Selection	<0,05	<0,05	0,117	<0,05	<0,05	<0,05	-	<0,05
Linq Parallel for Selection	<0,05	<0,05	<0,05	<0,05	0,518	<0,05	<0,05	-

Table A.65: Table showing the *p*-values for the group Numbrix Puzzle with regards to Elapsed Time.

Package Energy *p*-Values	Original	Linq	Parallel for	Selection	Linq Parallel for	Linq Selection	Parallel for Selection	Linq Parallel for Selection
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq	<0,05	-	<0,05	<0,05	<0,05	0,283	<0,05	<0,05
Parallel for	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Selection	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Linq Parallel for	<0,05	<0,05	<0,05	<0,05	-	<0,05	0,190	0,908
Linq Selection	<0,05	0,283	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for Selection	<0,05	<0,05	<0,05	<0,05	0,190	<0,05	-	0,211
Linq Parallel for Selection	<0,05	<0,05	<0,05	<0,05	0,908	<0,05	0,211	-

Table A.66: Table showing the *p*-values for the group Numbrix Puzzle with regards to Package Energy.

DRAM Energy *p*-Values	Original	Linq	Parallel for	Selection	Linq Parallel for	Linq Selection	Parallel for Selection	Linq Parallel for Selection
Original	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Parallel for	<0,05	<0,05	-	<0,05	<0,05	<0,05	0,262	<0,05
Selection	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Linq Parallel for	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,797
Linq Selection	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Parallel for Selection	<0,05	<0,05	0,262	<0,05	<0,05	<0,05	-	<0,05
Linq Parallel for Selection	<0,05	<0,05	<0,05	<0,05	0,797	<0,05	<0,05	-

Table A.67: Table showing the *p*-values for the group Numbrix Puzzle with regards to DRAM Energy.

A.3.8 Sum To 100

Elapsed Time *p*-Values	Original	No Linq Concurrent	Selection	Objects
Original	-	<0,05	<0,05	<0,05
No Linq Concurrent	<0,05	-	<0,05	<0,05
Selection	<0,05	<0,05	-	<0,05
Objects	<0,05	<0,05	<0,05	-
No Linq Concurrent Selection	<0,05	<0,05	<0,05	<0,05
No Linq Concurrent Objects	<0,05	<0,05	<0,05	<0,05
Selection Objects	<0,05	<0,05	<0,05	0,549
No Linq Concurrent Selection Objects	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	No Linq Concurrent Selection	No Linq Concurrent Objects	Selection Objects	No Linq Concurrent Selection Objects
Original	<0,05	<0,05	<0,05	<0,05
No Linq Concurrent	<0,05	<0,05	<0,05	<0,05
Selection	<0,05	<0,05	<0,05	<0,05
Objects	<0,05	<0,05	0,549	<0,05
No Linq Concurrent Selection	-	<0,05	<0,05	<0,05
No Linq Concurrent Objects	<0,05	-	<0,05	<0,05
Selection Objects	<0,05	<0,05	-	<0,05
No Linq Concurrent Selection Objects	<0,05	<0,05	<0,05	-

Table A.68: Table showing the *p*-values for the group Sum to Hundred with regards to Elapsed Time.

Package Energy *p*-Values	Original	No Linq Concurrent	Selection	Objects
Original	-	<0,05	<0,05	<0,05
No Linq Concurrent	<0,05	-	<0,05	<0,05
Selection	<0,05	<0,05	-	<0,05
Objects	<0,05	<0,05	<0,05	-
No Linq Concurrent Selection	<0,05	<0,05	<0,05	<0,05
No Linq Concurrent Objects	<0,05	<0,05	<0,05	<0,05
Selection Objects	<0,05	<0,05	<0,05	<0,05
No Linq Concurrent Selection Objects	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	No Linq Concurrent Selection	No Linq Concurrent Objects	Selection Objects	No Linq Concurrent Selection Objects
Original	<0,05	<0,05	<0,05	<0,05
No Linq Concurrent	<0,05	<0,05	<0,05	<0,05
Selection	<0,05	<0,05	<0,05	<0,05
Objects	<0,05	<0,05	<0,05	<0,05
No Linq Concurrent Selection	-	<0,05	<0,05	<0,05
No Linq Concurrent Objects	<0,05	-	<0,05	<0,05
Selection Objects	<0,05	<0,05	-	<0,05
No Linq Concurrent Selection Objects	<0,05	<0,05	<0,05	-

Table A.69: Table showing the *p*-values for the group Sum to Hundred with regards to Package Energy.

DRAM Energy *p*-Values	Original	No Linq Concurrent	Selection	Objects
Original	-	<0,05	<0,05	<0,05
No Linq Concurrent	<0,05	-	<0,05	<0,05
Selection	<0,05	<0,05	-	<0,05
Objects	<0,05	<0,05	<0,05	-
No Linq Concurrent Selection	<0,05	0,623	<0,05	<0,05
No Linq Concurrent Objects	<0,05	<0,05	<0,05	<0,05
Selection Objects	<0,05	<0,05	<0,05	0,344
No Linq Concurrent Selection Objects	<0,05	<0,05	<0,05	<0,05

DRAM Energy *p*-Values	No Linq Concurrent Selection	No Linq Concurrent Objects	Selection Objects	No Linq Concurrent Selection Objects
Original	<0,05	<0,05	<0,05	<0,05
No Linq Concurrent	0,623	<0,05	<0,05	<0,05
Selection	<0,05	<0,05	<0,05	<0,05
Objects	<0,05	<0,05	0,344	<0,05
No Linq Concurrent Selection	-	<0,05	<0,05	<0,05
No Linq Concurrent Objects	<0,05	-	<0,05	<0,05
Selection Objects	<0,05	<0,05	-	<0,05
No Linq Concurrent Selection Objects	<0,05	<0,05	<0,05	-

Table A.70: Table showing the *p*-values for the group Sum to Hundred with regards to DRAM Energy.

A.4 Variance of Old Benchmarks

As part of the ensuring that the framework is usable we have elected to test the variance of around 1% observed in [3]. To test this we have run the `Concurrent Dictionary Insertion` benchmark [69], which was used to establish the variance of results, 10 times. To specify this means that the benchmark is run with the required amount of iterations 10 times. We then take the results and calculate the variance of the results.

This work with variance of results from [3] is used in Section 3.3.

Concurrent Dictionary Insertion	Package Energy (μJ)
Run 1	723.219.893,02
Run 2	727.406.518,56
Run 3	730.810.145,79
Run 4	729.193.151,86
Run 5	729.521.008,30
Run 6	735.344.991,05
Run 7	735.675.430,30
Run 8	736.055.316,63
Run 9	729.714.777,17
Run 10	734.041.979,98
Percentage Variance	1,74%

Table A.71: Results and variance of the `ConcurrentDictionaryInsertion` benchmark from [3].

Looking at Table A.71 we can see two differences from what was found

in [3, p. 166]. The first and most important difference is the variance of 1,74%, which is different from the 1% from [3, p. 166]. This means that we cannot rely on the expected 1% variance nor the 2% difference needed to determine if one language construct is better than another. The second difference is the amount of energy consumed. These benchmarks consume approximately 35.000 to 45.000 (μJ) less than the approxiate median presented in [3, p. 166]. We believe this energy difference is due to changes in the framework itself. The framework used to use `int` for handling iterations in benchmarks, this is now changed to use `uint`, which was found to be more energy efficient than `int`. This change in datatype might also be the cause of different variance from earlier observations.

Based on this observation we decide to run all benchmarks from [3] again. They are run 10 times and from these numbers we then calculate all the benchmarks variance. We have found that most results vary with more then 1%. The lowest variance observed from is from the `Collections` group, specifically the `TableCopy` subgroup, where the `ConcurrentDictionaryCopyManualForeach` benchmark had a variance of 1,38%. The highest variance is 28,61%, which was observed for the `ClassFieldStatic` benchmark from the `ObjectFieldAccess` subgroup of `Objects`. On average the benchmarks have a variance of 7,57%. This means that we cannot us the 2% difference between language constructs to determine if one is better.

A.5 Microbenchmark Variance Tables

This section presents the variance of different microbenchmarks and is utilized in Section 4.1 and Section 4.2.

A.5.1 Concurrency

Package Energy (μJ)	Sequential	ParallelForDefault	ThreadPool1	UsingTasks	ManualThread4	ManualThread192
Run 1	26.316,458	147.401,744	3.559.783,970	8.784.535,228	18.486,501	27.524,666
Run 2	20.952,112	56.998,339	3.583.976,157	8.906.653,281	17.793,430	30.996,802
Run 3	27.463,649	147.896,987	3.697.594,809	9.054.771,099	20.012,925	29.925,783
Run 4	28.415,678	146.590,932	3.606.671,884	8.998.859,433	19.871,567	32.558,288
Run 5	21.146,669	148.239,190	3.743.934,497	8.780.267,808	19.744,738	27.918,195
Run 6	22.395,317	56.930,340	3.721.558,688	9.005.834,920	17.263,200	27.929,695
Run 7	22.392,180	55.334,487	3.522.992,744	8.965.410,760	20.140,061	119.907,111
Run 8	26.576,004	56.401,457	3.341.213,007	8.831.783,837	20.679,105	29.698,353
Run 9	27.333,060	56.946,280	3.487.245,938	8.962.240,371	20.925,487	31.760,979
Run 10	22.249,161	55.407,654	3.623.365,543	8.898.850,408	20.223,977	26.770,860
Percentage Variance	26,27%	62,67%	10,76%	3,03%	17,50%	77,67%

Table A.72: Results of running the Concurrency 1 benchmarks 10 times and percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Sequential	ParallelForDefault	ThreadPool10mil	UsingTasks	ManualThread4	ManualThread192
Run 1	91.923.502.878,289	66.172.046.205,357	84.702.283.208,225	66.595.998.104,977	66.585.275.000,000	94.444.796.875,000
Run 2	91.742.242.500,000	66.008.294.859,375	84.605.395.092,933	65.406.730.004,947	66.300.174.218,750	94.697.288.194,444
Run 3	90.862.408.653,846	66.232.315.952,846	87.033.102.212,311	66.492.112.658,074	67.303.442.471,591	97.429.026.041,667
Run 4	90.469.626.358,696	65.969.360.139,716	86.637.981.602,357	65.551.684.375,000	66.157.743.750,000	95.958.271.875,000
Run 5	92.419.660.714,286	65.220.966.929,201	86.784.266.285,644	65.570.962.583,149	65.830.442.968,750	95.568.819.196,429
Run 6	89.951.475.000,000	65.820.981.964,761	85.391.302.868,897	65.337.750.044,263	66.765.852.343,750	94.102.098.011,364
Run 7	91.232.066.666,667	66.561.620.127,540	86.002.825.028,767	64.841.130.642,361	67.002.125.000,000	94.340.459.375,000
Run 8	98.014.959.375,000	67.042.229.700,855	86.942.467.966,511	67.278.532.728,909	68.139.568.750,000	101.100.351.721,939
Run 9	97.315.403.273,810	66.995.395.008,411	73.514.787.777,549	66.855.840.840,242	68.756.403.906,250	98.003.973.030,822
Run 10	92.197.090.701,220	66.002.791.998,742	85.403.231.047,977	65.866.628.245,192	66.129.989.843,750	94.357.755.514,706
Percentage Variance	8,23%	2,72%	15,53%	3,62%	4,26%	6,92%

Table A.73: Results of running the Concurrency 10 mil benchmarks 10 times and percentage variance between highest and lowest recorded energy consumption.

A.5.2 Casting

Package Energy (μJ)	Implicit	Explicit
Run 1	11.918,392	11.481,019
Run 2	12.659,945	11.679,989
Run 3	11.487,479	11.282,674
Run 4	12.198,790	11.757,882
Run 5	12.352,257	12.012,479
Run 6	11.980,719	11.570,813
Run 7	10.513,979	11.918,606
Run 8	10.765,185	12.132,747
Run 9	10.449,613	11.818,791
Run 10	12.147,003	11.575,550
Mean	11.647,336	11.723,055
Percentage Variance	17,46%	7,01%

Table A.74: Results of running the numeric casting benchmarks 10 times and percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Implicit	Explicit
Run 1	9.960,346	11.087,733
Run 2	12.150,015	10.880,777
Run 3	10.028,762	10.756,345
Run 4	11.730,307	10.699,720
Run 5	11.661,608	10.662,476
Run 6	11.941,362	10.735,334
Run 7	9.376,421	9.592,079
Run 8	10.133,534	10.840,225
Run 9	11.805,946	10.772,480
Run 10	10.105,341	10.717,139
Mean	10.674,431	10.889,364
Percentage Variance	22,83%	13,49%

Table A.75: Results of running the numeric casting benchmarks 10 times without garbage collection and the percentage variance between highest and lowest recorded energy consumption.

A.5.3 Parameter Mechanisms

Package Energy (μJ)	In	Ref
Run 1	4.431,093	4.511,823
Run 2	4.933,167	4.945,240
Run 3	4.510,980	4.425,249
Run 4	4.933,359	4.922,503
Run 5	4.584,889	4.461,174
Run 6	4.892,204	4.931,325
Run 7	4.920,479	4.934,951
Run 8	4.934,406	4.937,202
Run 9	4.905,588	4.931,700
Run 10	4.727,539	4.860,801
Mean	4.777,370	4.786,197
Percentage Variance	10,20%	10,51%

Table A.76: Results of running the parameter mechanism benchmarks 10 times for the non-modifying group and percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	In	Ref
Run 1	4.386,973	4.389,161
Run 2	4.361,122	4.292,611
Run 3	4.404,131	4.380,547
Run 4	4.377,390	4.353,308
Run 5	4.545,227	4.458,559
Run 6	4.397,544	4.354,786
Run 7	4.406,495	4.395,204
Run 8	4.547,304	4.485,936
Run 9	4.441,854	4.372,908
Run 10	4.356,289	4.342,428
Mean	4.422,433	4.382,544
Percentage Variance	4,20%	4,31%

Table A.77: Results of running the parameter mechanism benchmarks 10 times for the non-modifying group without garbage collection and percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Ref	Out
Run 1	9.463,114	10.325,924
Run 2	9.125,903	9.478,871
Run 3	10.300,671	9.202,392
Run 4	9.678,844	10.378,214
Run 5	9.059,249	9.115,433
Run 6	10.538,125	10.585,574
Run 7	9.213,548	9.214,998
Run 8	9.330,302	9.124,187
Run 9	9.123,823	9.128,848
Run 10	9.072,312	9.104,769
Mean	9.490,589	9.565,921
Percentage Variance	14,03%	13,99%

Table A.78: Results of running the parameter mechanism benchmarks 10 times for the modifying group and percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Ref	Out
Run 1	9.885,498	9.980,910
Run 2	9.543,848	8.627,433
Run 3	10.088,796	9.901,853
Run 4	9.515,459	9.304,927
Run 5	9.747,960	10.214,557
Run 6	9.348,887	9.926,074
Run 7	8.959,909	8.562,990
Run 8	10.158,331	9.293,846
Run 9	10.413,265	10.202,658
Run 10	9.534,553	9.448,829
Mean	9.719,650	9.546,408
Percentage Variance	13,96%	16,17%

Table A.79: Results of running the parameter mechanism benchmarks 10 times for the modifying group without garbage collection and percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Ref Small	Ref Large	Return Small	Return Large
Run 1	12.012,929	12.327,188	21.769,994	33.236,148
Run 2	13.352,125	12.963,082	21.917,562	33.342,868
Run 3	13.150,100	13.141,350	21.861,012	33.091,869
Run 4	12.379,328	12.912,109	21.832,728	32.917,257
Run 5	12.720,021	12.511,503	21.880,613	33.044,768
Run 6	13.100,987	13.631,337	22.352,039	33.905,060
Run 7	12.832,459	12.934,406	22.014,195	32.969,680
Run 8	13.027,501	13.424,826	21.950,939	33.448,065
Run 9	14.045,794	13.717,367	24.342,700	36.803,710
Run 10	12.737,094	12.690,040	21.829,683	32.744,363
Mean	12.935,834	13.025,320	22.175,147	33.550,379
Percentage Variance	14,73%	10,13%	10,57%	11,03%

Table A.80: Results of running the parameter mechanism benchmarks 10 times for the sub-group that looks at modifying value types and percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Out Return	Ref Return	Return	Return Alt
Run 1	6.600,527	6.573,697	6.636,782	6.633,167
Run 2	6.558,861	6.708,636	6.550,264	6.787,053
Run 3	6.541,251	6.577,130	6.468,599	6.479,355
Run 4	7.156,834	7.076,353	7.108,444	7.159,310
Run 5	6.341,026	6.433,188	6.360,542	6.566,057
Run 6	7.040,034	7.327,155	6.917,578	6.759,295
Run 7	6.916,205	7.257,501	7.006,375	6.811,492
Run 8	7.111,002	7.178,995	6.942,417	6.787,071
Run 9	7.197,149	6.836,232	6.922,463	7.058,714
Run 10	6.956,918	6.754,603	6.895,078	7.018,574
Mean	6.841,981	6.872,349	6.780,854	6.806,009
Percentage Variance	11,90%	12,20%	10,52%	9,50%

Table A.81: Results of running the parameter mechanism benchmarks 10 times for the sub-group that looks at returning values and percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	RefReturn	Return	ReturnAlt	OutReturn
Run 1	6.413,84	6.503,30	6.470,03	6.493,43
Run 2	6.153,83	6.209,67	6.317,65	5.980,40
Run 3	6.535,72	6.454,57	6.412,16	6.412,28
Run 4	6.519,27	6.486,77	6.483,21	6.467,82
Run 5	6.330,47	6.418,52	6.333,05	6.269,64
Run 6	6.262,69	6.047,79	5.818,08	6.321,18
Run 7	6.252,35	6.432,98	6.439,84	6.379,68
Run 8	6.245,88	6.309,57	6.152,59	6.265,14
Run 9	6.175,02	5.953,82	6.061,80	6.149,97
Run 10	6.193,03	6.126,05	6.192,85	6.009,07
Mean	6.308,21	6.294,30	6.268,13	6.274,86
Percentage variance	5,84%	8,45%	10,26%	7,90%

Table A.82: Results of running the parameter mechanism benchmarks 10 times without garbage collection for the sub-group that looks at returning values and percentage variance between highest and lowest recorded energy consumption.

A.5.4 Lambda

Package Energy (μJ)	Lambda Delegate Closure	Lambda	Lambda Action	Lambda Delegate	Lambda Closure	Lambda Param
Run 1	24.837,101	17.660,982	19.516,888	20.405,484	21.063,905	18.018,340
Run 2	24.782,211	23.373,991	19.502,404	20.469,020	18.828,809	17.955,380
Run 3	24.778,117	23.159,183	19.517,638	20.750,273	21.377,635	17.949,651
Run 4	24.642,696	17.810,134	19.515,296	20.272,919	21.414,921	20.632,075
Run 5	24.737,782	17.841,451	19.527,915	20.657,954	21.131,600	20.610,353
Run 6	24.514,584	22.720,303	19.387,149	19.675,268	18.099,626	19.911,644
Run 7	24.870,659	17.932,074	19.414,047	20.398,249	18.871,400	17.923,593
Run 8	18.968,954	17.959,564	21.151,444	20.654,700	19.178,931	20.602,909
Run 9	18.532,958	17.924,481	20.956,728	20.352,725	18.634,180	17.917,159
Run 10	23.340,546	22.633,673	18.932,903	19.334,558	20.954,250	19.567,148
Mean	23.400,561	19.901,584	19.742,241	20.297,115	19.955,526	19.108,825
Percentage Variance	25,48%	24,44%	10,49%	6,82%	15,48%	13,16%

Table A.83: Results of running the lambda benchmarks 10 times and percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Lambda Delegate Closure	Lambda	Lambda Action	Lambda Delegate	Lambda Closure	Lambda Param
Run 1	20.882,582	19.866,112	19.235,355	20.226,798	24.284,908	20.221,895
Run 2	18.960,924	20.253,897	19.229,624	17.721,253	18.612,461	22.630,833
Run 3	18.761,230	20.546,229	24.209,110	17.751,312	18.949,888	20.494,368
Run 4	20.810,754	17.814,974	19.056,980	20.272,919	18.430,539	20.066,350
Run 5	18.730,889	20.277,607	19.324,575	20.431,687	24.568,799	20.398,284
Run 6	18.008,200	17.786,923	19.136,118	17.699,631	18.284,277	21.804,056
Run 7	20.957,708	17.774,697	24.216,443	20.485,613	18.252,682	20.151,735
Run 8	20.731,674	20.088,074	19.294,746	20.055,281	24.455,202	20.017,348
Run 9	18.190,350	19.957,138	24.298,088	19.960,647	24.515,049	19.576,395
Run 10	20.832,400	19.680,331	18.845,707	19.817,752	24.169,850	20.149,576
Mean	19.686,671	19.404,598	20.684,675	19.442,289	21.452,366	20.551,084
Percentage Variance	14,07%	13,49%	22,44%	13,60%	25,71%	13,50%

Table A.84: Results of running the lambda benchmarks 10 times without garbage collection and the percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Lambda Delegate Closure	Lambda	Lambda Action	Lambda Delegate	Lambda Closure	Lambda Param
Run 1	22.578,603	24.071,164	20.315,260	23.633,569	18.531,868	23.614,288
Run 2	24.536,012	24.464,572	19.188,061	23.758,754	21.044,145	23.886,953
Run 3	24.564,251	25.924,479	19.288,741	26.313,364	18.951,775	23.890,053
Run 4	19.227,260	26.496,383	21.411,619	24.243,789	21.501,688	24.076,113
Run 5	19.120,361	23.749,802	21.649,350	26.231,191	19.225,532	26.454,723
Run 6	19.370,844	23.885,998	21.906,391	24.267,216	19.148,339	24.325,355
Run 7	18.753,067	23.798,145	21.176,855	23.808,540	19.171,053	26.343,937
Run 8	24.692,491	26.237,335	19.391,491	23.763,535	19.169,053	23.666,447
Run 9	18.366,397	23.745,668	20.804,593	23.649,666	20.876,882	23.698,462
Run 10	18.966,143	25.891,424	19.383,851	26.135,628	18.830,562	23.754,667
Mean	21.017,543	24.826,497	20.451,621	24.580,525	19.645,090	24.371,010
Percentage Variance	25,62%	10.38%	12.40%	10.18%	13.81%	10,74%

Table A.85: Results of running variations of the lambda benchmarks that do not use lambda 10 times and the percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Lambda
Run 1	16.864,931
Run 2	17.735,701
Run 3	21.939,734
Run 4	22.158,489
Run 5	17.819,310
Run 6	22.630,898
Run 7	22.996,795
Run 8	23.050,470
Run 9	22.456,042
Run 10	23.445,831
Mean	21.109,820
Percentage Variance	28,07%

Table A.86: Results of running variations of only the `lambda` benchmark 10 times and the percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Lambda
Run 1	18.848,425
Run 2	18.884,321
Run 3	20.098,159
Run 4	19.503,966
Run 5	20.345,266
Run 6	20.691,180
Run 7	19.958,291
Run 8	17.795,639
Run 9	20.421,153
Run 10	20.236,796
Mean	19.678,796
Percentage Variance	13,99%

Table A.87: Results of running variations of only the `lambda` benchmark 10 times without garbage collection and the percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Lambda
Run 1	16.394,609
Run 2	17.769,264
Run 3	17.558,884
Run 4	18.050,004
Run 5	17.834,425
Run 6	17.814,629
Run 7	17.853,952
Run 8	18.064,057
Run 9	17.829,745
Run 10	17.817,050
Mean	17.698,662
Percentage Variance	9,24%

Table A.88: Results of running variations of only the `lambda` benchmark 10 times with R2R compilation and the percentage variance between highest and lowest recorded energy consumption.

Package Energy (μJ)	Lambda
Run 1	16.269,520
Run 2	17.856,171
Run 3	16.601,412
Run 4	17.830,920
Run 5	17.792,188
Run 6	17.818,466
Run 7	17.822,885
Run 8	17.923,736
Run 9	17.985,268
Run 10	18.027,333
Mean	17.592,790
Percentage Variance	9,75%

Table A.89: Results of running variations of only the `lambda` benchmark 10 times with R2R compilation and garbage collection turned off and the percentage variance between highest and lowest recorded energy consumption.

A.6 Macrobenchmark Variance Tables

This section presents the variance of different macrobenchmarks and is utilized in Section 5.2 and Section 5.3.

A.6.1 100 Prisoners

Package Energy (μJ)	ParallelForArrayNoLINQ	ParallelFor	ParallelForArray	ArrayNoLINQ
Run 1	12.538.147.692.307,69	25.907.526.500.000,00	25.876.201.384.615,38	15.275.737.026.315,79
Run 2	12.643.839.095.238,10	25.725.715.400.000,00	26.278.399.500.000,00	14.947.654.647.058,82
Run 3	12.631.957.565.217,39	26.102.463.500.000,00	26.201.740.583.333,33	15.230.448.347.826,09
Run 4	12.672.594.000.000,00	25.679.243.300.000,00	26.359.576.100.000,00	15.190.801.692.307,69
Run 5	12.706.510.400.000,00	25.858.328.133.333,33	26.135.028.300.000,00	15.186.240.440.000,00
Run 6	12.435.901.583.333,33	25.800.410.100.000,00	25.659.520.916.666,67	14.798.991.800.000,00
Run 7	12.295.238.300.000,00	25.742.626.454.545,45	25.388.075.900.000,00	15.331.269.708.333,33
Run 8	12.540.708.800.000,00	25.770.289.300.000,00	25.959.125.400.000,00	14.852.684.100.000,00
Run 9	12.373.173.900.000,00	25.716.731.090.909,09	25.545.662.200.000,00	15.471.475.300.000,00
Run 10	12.554.057.250.000,00	25.642.164.600.000,00	26.091.953.833.333,33	14.866.347.586.206,90
Mean	12.539.212.858.609,60	25.794.549.837.878,80	25.949.528.411.794,90	15.115.165.064.804,80
Percentage variance	3,24%	1,76%	3,69%	4,35%

Package Energy (μJ)	ParallelForNoLINQ	NoLINQ	Array	Original
Run 1	12.896.718.761.904,76	16.007.505.571.428,57	31.462.627.000.000,00	31.538.132.250.000,00
Run 2	13.059.397.428.571,43	15.599.768.413.043,48	30.823.621.000.000,00	31.495.999.869.565,22
Run 3	13.012.356.846.153,85	15.647.428.058.823,53	31.085.070.600.000,00	31.792.860.782.608,70
Run 4	13.173.293.564.102,56	15.850.044.448.275,86	30.690.992.000.000,00	31.845.359.529.411,77
Run 5	13.185.912.636.363,64	15.846.011.697.674,42	30.893.194.900.000,00	31.742.061.848.484,85
Run 6	12.601.965.818.181,82	16.303.828.000.000,00	30.886.053.600.000,00	32.221.902.300.000,00
Run 7	12.434.913.000.000,00	15.536.172.694.444,45	31.189.226.800.000,00	31.480.266.200.000,00
Run 8	13.047.853.750.000,00	15.756.974.333.333,33	30.890.157.545.454,55	31.559.544.400.000,00
Run 9	12.650.643.000.000,00	15.878.150.200.000,00	31.304.596.000.000,00	31.592.472.800.000,00
Run 10	13.126.060.000.000,00	15.794.505.820.512,82	31.094.780.500.000,00	31.420.982.800.000,00
Mean	12.918.911.480.527,80	15.822.038.923.753,60	31.032.031.994.545,40	31.668.958.278.007,00
Percentage variance	5,70%	4,71%	2,45%	2,49%

Table A.90: Results of running the 100 Prisoner benchmarks 10 times and the percentage variance between highest and lowest recorded energy consumption.

A.6.2 Almost Prime

Package Energy (μJ)	ManualThreadArrayStruct	Array	ManualThreadArray	ArrayStruct
Run 1	263.821.768.995,10	9.482.240.234,38	258.866.513.849,43	9.846.781.445,31
Run 2	273.560.141.927,08	9.511.112.695,31	269.255.902.343,75	10.247.806.589,23
Run 3	259.294.742.424,24	9.537.457.812,50	255.210.535.447,76	9.762.548.242,19
Run 4	259.300.113.281,25	9.393.590.877,76	261.350.136.363,64	9.644.602.929,69
Run 5	265.973.595.170,46	9.520.542.187,50	265.233.337.725,90	9.795.128.125,00
Run 6	264.231.702.891,79	9.486.495.898,44	266.950.598.651,96	9.488.168.501,42
Run 7	259.518.314.338,24	9.756.232.096,35	263.695.677.827,38	9.773.189.615,89
Run 8	270.151.268.269,23	9.564.952.974,76	266.016.270.188,05	9.823.070.703,13
Run 9	268.311.948.908,73	9.583.668.870,19	265.588.059.375,00	9.530.925.195,31
Run 10	265.755.223.684,21	9.900.675.585,94	270.932.284.090,91	9.815.882.226,56
Mean	264.991.881.989,03	9.573.696.923,31	264.309.931.586,38	9.772.810.357,37
Percentage variance	5,21%	5,12%	5,80%	7,41%

Package Energy (μJ)	Original	Struct	ManualThreadStruct	ManualThread
Run 1	9.180.029.557,29	9.457.325.390,63	259.637.167.564,66	263.587.480.416,67
Run 2	9.855.757.235,44	9.800.289.843,75	272.299.532.196,97	265.975.711.538,46
Run 3	9.626.522.832,96	9.725.223.828,13	255.089.994.612,07	263.264.270.833,33
Run 4	9.399.086.618,13	9.409.110.795,46	260.001.626.536,89	260.367.370.703,13
Run 5	9.628.748.046,88	9.902.940.609,98	265.975.137.609,65	259.318.545.094,94
Run 6	9.826.831.054,69	9.851.501.953,13	268.097.438.858,70	265.485.277.542,37
Run 7	9.461.420.176,63	9.455.871.289,06	263.945.186.342,59	261.066.669.763,51
Run 8	9.787.713.169,64	9.903.453.125,00	267.191.482.421,88	264.735.292.338,71
Run 9	9.709.413.020,83	9.893.964.257,81	266.896.474.563,95	261.818.011.938,20
Run 10	9.813.023.615,06	10.025.746.508,05	269.834.671.274,04	266.524.500.932,84
Mean	9.628.854.532,76	9.742.542.760,10	264.896.871.198,14	263.214.313.110,22
Percentage variance	6,86%	6,15%	6,32%	2,70%

Table A.91: Results of running the Almost Prime benchmarks 10 times and the percentage variance between highest and lowest recorded energy consumption.

A.6.3 Chebyshev

Package Energy (μJ)	Original	Array	Datatype	Concurrent
Run 1	49.178.185.105.263,10	45.950.765.105.263,10	49.352.868.805.555,50	59.392.086.333.333,30
Run 2	51.310.030.900.000,00	46.745.363.636.363,60	50.900.486.806.451,60	60.420.758.440.000,00
Run 3	46.769.783.900.000,00	46.519.412.200.000,00	47.930.673.583.333,30	60.046.398.666.666,60
Run 4	47.315.497.800.000,00	44.618.064.315.789,40	47.971.745.000.000,00	59.202.134.250.000,00
Run 5	50.720.673.400.000,00	46.096.940.100.000,00	49.581.358.444.444,40	59.790.862.500.000,00
Run 6	48.675.994.875.000,00	46.472.758.073.529,40	50.456.315.230.769,20	59.820.827.235.294,10
Run 7	46.945.351.000.000,00	45.330.042.218.750,00	48.684.055.100.000,00	58.992.860.600.000,00
Run 8	49.113.954.545.454,50	46.448.556.600.000,00	49.147.365.600.000,00	59.950.220.933.333,30
Run 9	47.777.393.125.000,00	46.432.016.100.000,00	49.387.627.629.629,60	60.167.001.000.000,00
Run 10	47.535.673.076.923,00	45.548.669.736.842,10	47.476.795.000.000,00	59.653.132.000.000,00
Mean	48.534.253.772.764,10	46.016.258.808.653,80	49.088.929.120.018,40	59.743.628.195.862,70
Percentage variance	8,85%	4,55%	6,73%	2,36%

Package Energy (μJ)	ArrayDatatype	ConcurrentDatatype	ConcurrentArray	ConcurrentArrayDatatype
Run 1	47.209.912.272.727,20	60.218.567.615.384,60	57.172.311.272.727,20	57.870.285.272.727,20
Run 2	47.447.986.500.000,00	60.468.571.333.333,30	57.974.600.710.526,30	58.165.454.700.000,00
Run 3	46.617.123.100.000,00	59.704.691.911.764,70	58.797.579.000.000,00	57.597.367.942.307,60
Run 4	47.386.222.827.586,20	59.954.015.200.000,00	57.969.268.866.666,60	56.929.181.700.000,00
Run 5	46.867.543.600.000,00	59.401.642.307.692,30	58.016.315.750.000,00	58.068.843.272.727,20
Run 6	46.772.011.500.000,00	60.909.218.476.190,40	58.113.389.421.052,60	58.137.058.300.000,00
Run 7	46.998.408.700.000,00	59.710.690.692.307,60	57.695.537.200.000,00	58.233.017.400.000,00
Run 8	46.916.097.000.000,00	59.704.730.000.000,00	57.436.479.769.230,70	57.536.846.200.000,00
Run 9	46.389.981.300.000,00	60.212.107.800.000,00	57.721.169.800.000,00	57.782.115.300.000,00
Run 10	46.746.517.863.636,30	60.299.420.307.692,30	57.674.925.962.962,90	57.395.983.400.000,00
Mean	46.935.180.466.395,00	60.058.365.564.436,50	57.857.157.775.316,60	57.771.615.348.776,20
Percentage variance	2,23%	2,48%	2,76%	2,24%

Table A.92: Results of running the Chebyshev benchmarks 10 times and the percentage variance between highest and lowest recorded energy consumption.

A.6.4 Compare length of two strings

Package Energy (μJ)	Original	Selection	ParallelFor1	ParallelFor2
Run 1	8.518.237,45	8.480.392,08	58.759.381,10	84.681.729,08
Run 2	8.696.144,10	8.756.947,52	57.827.433,45	79.652.463,28
Run 3	8.413.528,49	8.320.710,42	58.458.606,30	82.950.555,17
Run 4	8.113.116,32	8.127.725,31	57.564.155,58	83.212.352,75
Run 5	8.653.349,88	8.642.977,33	57.890.824,64	80.880.890,53
Run 6	8.754.980,28	8.493.884,85	57.987.920,13	82.395.449,32
Run 7	8.320.705,65	8.247.170,29	58.958.481,60	85.389.061,61
Run 8	8.531.918,48	8.415.863,42	59.183.721,16	85.249.192,37
Run 9	8.570.543,86	8.655.212,59	58.438.778,47	80.019.593,56
Run 10	8.720.765,69	8.786.202,62	58.411.657,72	90.278.049,97
Mean	8.529.329,019	8.492.708,643	58.348.096,013	83.470.933,764
Percentage variance	7,33%	7,49%	2,74%	11,77%

Package Energy (μJ)	Foreach	SelectionForeach	ParallelFor1Foreach	ParallelFor2Foreach
Run 1	12.308.974,37	12.281.944,77	63.085.587,72	88.356.208,80
Run 2	12.579.120,64	12.556.506,67	62.080.914,58	83.815.324,40
Run 3	12.169.882,62	12.208.479,37	62.500.292,97	85.857.244,11
Run 4	11.956.378,76	11.998.126,54	63.571.125,03	88.110.237,12
Run 5	12.917.375,76	12.788.353,54	62.805.998,23	85.148.323,06
Run 6	12.700.576,40	12.690.122,41	63.148.621,10	85.907.768,25
Run 7	12.526.780,32	12.790.269,22	63.470.169,07	89.185.530,03
Run 8	12.243.421,03	12.349.027,76	63.519.947,82	88.588.020,33
Run 9	12.345.603,94	12.370.831,11	62.945.789,34	84.491.184,24
Run 10	12.925.979,23	12.793.126,49	64.420.721,44	89.568.665,35
Mean	12.467.409,31	12.482.678,79	63.154.916,73	86.902.850,57
Percentage variance	7,50%	6,21%	3,63%	6,42%

Package Energy (μJ)	SelectionParallelFor1	SelectionParallelFor2	SelectionParallelFor1Foreach	SelectionParallelFor2Foreach
Run 1	58.143.856,05	86.340.315,80	63.555.351,80	87.879.838,94
Run 2	57.620.060,31	79.050.073,38	62.330.753,03	83.884.195,71
Run 3	58.278.944,40	82.759.328,12	63.016.383,70	86.078.618,62
Run 4	57.919.733,43	85.637.002,30	63.105.417,63	87.392.665,10
Run 5	58.511.089,33	81.540.435,03	63.022.703,99	85.707.209,02
Run 6	57.943.064,12	82.998.649,60	63.636.270,91	86.048.259,31
Run 7	58.540.644,84	88.969.416,48	63.760.555,27	89.384.118,30
Run 8	59.020.973,21	85.541.716,84	64.034.037,02	89.536.704,25
Run 9	58.073.002,77	79.730.744,80	62.703.925,32	84.790.230,56
Run 10	59.282.767,49	89.754.018,55	64.230.452,73	90.922.708,40
Mean	58.333.413,59	84.232.170,09	63.339.585,14	87.162.454,82
Percentage variance	2,80%	11,93%	2,96%	7,74%

Table A.93: Results of running the Compare length of two strings benchmarks 10 times and the percentage variance between highest and lowest recorded energy consumption.

A.6.5 Dijkstra

Package Energy (μJ)	Original	NoLinq	Objects	Array
Run 1	61.842.655,83	46.746.571,60	62.695.472,13	61.662.761,26
Run 2	63.214.288,98	47.506.643,02	64.736.065,67	63.561.018,37
Run 3	62.871.068,32	47.470.787,05	64.475.761,41	63.831.821,99
Run 4	62.512.624,57	47.021.286,58	63.976.107,79	63.352.218,63
Run 5	63.559.027,71	47.490.856,93	63.031.471,25	63.112.725,34
Run 6	63.446.418,76	48.348.742,68	63.207.862,85	63.686.638,57
Run 7	63.061.551,92	46.888.719,37	65.070.689,39	63.040.037,54
Run 8	61.303.135,27	46.734.952,89	63.216.244,51	63.146.862,79
Run 9	62.791.354,18	46.611.732,48	62.580.897,52	62.708.333,33
Run 10	63.189.417,22	47.786.457,83	64.344.911,19	64.644.331,36
Mean	62.779.154,28	47.260.675,04	63.733.548,37	63.274.674,92
Percentage variance	3,55%	3,59%	3,83%	4,61%

Package Energy (μJ)	ArrayObjects	NoLinqArray	NoLinqObjects	NoLinqArrayObjects
Run 1	69.349.826,05	46.246.371,61	48.017.565,26	46.333.644,64
Run 2	63.446.897,96	46.684.426,88	48.005.831,15	45.725.285,12
Run 3	63.170.549,06	46.741.970,70	46.827.484,13	45.406.860,02
Run 4	63.991.476,44	47.757.808,69	47.491.136,93	46.352.701,04
Run 5	63.615.968,32	46.820.072,94	48.169.789,89	46.810.340,88
Run 6	62.651.093,92	47.098.539,73	47.419.564,68	46.411.177,06
Run 7	64.445.961,00	46.547.459,46	47.838.996,89	45.760.283,88
Run 8	67.756.198,12	46.951.265,69	48.871.408,64	45.811.947,66
Run 9	62.834.621,85	45.142.331,72	47.078.962,40	46.354.178,66
Run 10	63.684.046,94	46.198.837,28	48.269.813,54	46.212.759,40
Mean	64.494.663,97	46.618.908,47	47.799.055,35	46.117.917,84
Percentage variance	9,66%	5,48%	4,18%	3,00%

Table A.94: Results of running the Dijkstra benchmarks 10 times and the percentage variance between highest and lowest recorded energy consumption.

A.6.6 Four Square Puzzle

Package Energy (μJ)	Original	Selection	ParallelFor	NoLINQ
Run 1	23.446.142.187,50	23.836.241.796,88	30.749.800.644,19	14.812.726.562,50
Run 2	23.867.856.250,00	24.419.020.182,29	30.860.487.137,00	14.941.983.028,02
Run 3	23.439.419.140,63	23.850.165.234,38	30.845.266.715,94	14.513.622.265,63
Run 4	23.357.784.375,00	23.842.655.468,75	30.506.554.177,30	14.679.800.000,00
Run 5	23.518.096.484,38	24.231.510.904,95	30.972.767.510,32	14.845.714.453,13
Run 6	23.629.438.281,25	24.440.858.774,04	31.256.475.810,46	14.957.479.003,91
Run 7	23.252.666.015,63	23.746.834.765,63	30.361.560.243,80	14.531.408.203,13
Run 8	23.394.858.593,75	23.767.267.578,13	30.834.965.095,77	14.668.737.109,38
Run 9	23.656.784.765,63	23.850.546.484,38	31.208.162.819,60	14.782.042.367,79
Run 10	23.432.051.953,13	24.013.910.546,88	30.891.211.197,92	14.805.651.611,33
Mean	23.499.509.804,69	23.999.901.173,63	30.848.725.135,23	14.753.916.460,48
Percentage variance	2,58%	2,84%	2,86%	2,97%

Package Energy (μJ)	ParallelForSelection	ParallelForNoLINQ	NoLINQSelection	ParallelForNoLINQSelection
Run 1	30.680.783.145,34	19.344.332.730,75	14.527.256.059,70	19.233.176.847,68
Run 2	31.158.205.903,94	19.601.516.381,05	15.042.290.234,38	23.347.497.564,94
Run 3	31.221.182.207,31	19.208.850.395,70	14.584.694.921,88	19.459.468.922,84
Run 4	30.885.168.467,42	19.220.829.321,63	14.943.275.781,25	19.515.813.340,44
Run 5	31.027.052.984,02	19.322.852.856,74	14.742.432.421,88	19.377.279.202,06
Run 6	30.839.919.140,63	19.310.995.332,53	14.691.864.062,50	19.329.754.775,79
Run 7	31.009.556.773,79	19.097.596.749,83	14.729.152.343,75	19.399.927.882,54
Run 8	31.210.271.284,45	19.293.629.255,02	14.996.736.391,13	19.413.522.556,21
Run 9	30.985.839.621,80	19.397.046.585,08	14.819.389.204,55	19.518.850.784,56
Run 10	31.334.270.046,03	19.481.810.691,07	14.866.862.109,38	19.470.179.472,11
Mean	31.035.224.957,47	19.327.946.029,94	14.794.395.353,04	19.806.547.134,92
Percentage variance	2,09%	2,57%	3,42%	17,62%

Table A.95: Results of running the Four Square Puzzle benchmarks 10 times and the percentage variance between highest and lowest recorded energy consumption.

A.6.7 Numbrix

Package Energy (μJ)	Original	Linq	LinqParallelFor	Selection
Run 1	832.330.566,41	789.008.496,09	2.169.799.013,67	826.034.814,45
Run 2	893.496.655,27	826.935.958,86	2.186.510.187,32	844.601.247,15
Run 3	843.236.328,13	805.246.240,23	2.157.816.748,05	853.070.224,61
Run 4	841.995.898,44	786.636.254,88	2.239.213.725,84	831.679.345,70
Run 5	836.706.982,42	790.827.929,69	2.187.472.678,45	840.964.257,81
Run 6	842.950.724,28	789.081.542,97	2.164.698.071,29	847.109.448,24
Run 7	831.761.328,13	787.074.316,41	2.170.707.690,43	836.230.175,78
Run 8	847.215.277,78	800.348.402,24	2.091.000.776,81	832.846.069,34
Run 9	886.515.712,19	822.720.805,66	2.172.655.257,16	857.517.919,92
Run 10	884.352.922,71	833.706.917,32	2.206.267.045,46	851.311.802,46
Mean	854.056.239,58	803.158.686,44	2.174.614.119,45	842.136.530,55
Percentage variance	6,91%	5,65%	6,62%	3,67%

Package Energy (μJ)	ParallelFor	LinqSelection	ParallelForSelection	LinqParallelForSelection
Run 1	2.129.312.233,67	790.740.014,65	2.157.190.162,30	2.166.614.331,06
Run 2	2.164.787.451,17	787.223.339,84	2.178.385.951,45	2.172.221.130,37
Run 3	2.143.668.115,23	793.092.895,51	2.172.921.997,07	2.168.104.443,36
Run 4	2.241.368.798,83	790.328.759,77	2.241.836.214,19	2.252.893.337,67
Run 5	2.190.850.161,90	801.924.272,02	2.166.977.482,72	2.152.625.170,90
Run 6	2.178.940.551,76	790.561.206,06	2.163.838.134,77	2.182.606.071,92
Run 7	2.198.478.690,01	792.863.427,73	2.197.055.859,38	2.173.488.052,37
Run 8	2.096.534.232,76	793.048.242,19	2.069.286.181,64	2.076.319.852,94
Run 9	2.160.509.350,59	802.727.978,52	2.155.273.897,06	2.149.871.927,90
Run 10	2.200.970.382,69	797.134.858,63	2.207.314.186,79	2.181.780.712,89
Mean	2.170.541.996,86	793.964.499,49	2.171.008.006,74	2.167.652.503,14
Percentage variance	6,46%	1,93%	7,70%	7,84%

Table A.96: Results of running the Numbrix benchmarks 10 times and the percentage variance between highest and lowest recorded energy consumption.

A.6.8 Sum To Hundred

Package Energy (μJ)	Original	Objects	Selection	NoLinqConcurrent
Run 1	91.651.333.850,77	82.198.099.272,63	91.313.060.486,58	84.788.018.251,81
Run 2	93.136.945.140,86	82.463.216.282,90	91.360.144.300,21	88.358.789.852,94
Run 3	90.027.472.104,78	82.891.472.138,98	92.228.112.471,48	86.214.010.850,16
Run 4	90.824.146.870,20	82.815.633.365,60	91.374.246.969,20	84.671.352.540,47
Run 5	92.550.463.992,17	82.166.831.301,11	91.604.440.716,35	85.853.566.880,62
Run 6	91.161.759.220,16	82.705.282.142,86	90.584.423.253,68	84.139.999.518,00
Run 7	91.396.922.528,58	82.920.541.140,70	93.057.448.932,05	86.844.688.230,99
Run 8	90.745.594.063,76	82.332.707.913,31	91.730.280.060,92	84.676.083.824,38
Run 9	91.851.978.882,83	83.300.506.155,30	91.959.116.120,49	86.320.381.169,98
Run 10	92.704.300.974,98	83.940.324.558,42	92.621.469.348,31	86.699.526.756,54
Mean	91.605.091.762,91	82.773.461.427,18	91.783.274.265,93	85.856.641.787,59
Percentage variance	3,34%	2,11%	2,66%	4,77%

Package Energy (μJ)	NoLinqConcurrentSelection	NoLinqConcurrentObjects	SelectionObjects	NoLinqConcurrentSelectionObjects
Run 1	85.191.762.348,40	78.937.738.415,24	82.962.779.845,03	76.287.187.411,72
Run 2	86.023.896.584,82	78.994.622.301,14	83.943.059.226,93	77.922.144.577,21
Run 3	86.447.378.579,61	76.575.145.864,52	81.274.392.048,33	74.942.386.801,86
Run 4	86.166.693.621,96	78.110.929.455,45	83.558.020.975,20	76.988.152.721,77
Run 5	86.011.110.098,38	78.452.840.375,00	83.538.463.379,99	77.386.259.726,82
Run 6	82.354.493.323,44	76.462.545.209,39	82.405.852.557,35	77.282.411.188,47
Run 7	86.668.628.849,64	77.973.869.393,69	82.386.901.929,91	77.967.644.164,37
Run 8	85.467.298.814,35	78.273.400.074,41	83.017.973.198,79	78.661.043.242,87
Run 9	86.477.532.120,67	77.233.178.067,40	84.087.299.026,95	76.851.210.233,67
Run 10	87.194.442.165,42	79.204.853.872,51	83.921.996.759,50	79.084.224.644,89
Mean	85.800.323.650,67	78.021.912.302,87	83.109.673.894,80	77.337.266.471,37
Percentage variance	5,55%	3,46%	3,35%	5,24%

Table A.97: Results of running the Sum To Hundred benchmarks 10 times and the percentage variance between highest and lowest recorded energy consumption.

A.7 Concurrency Results

This section contains results for all of the different concurrency benchmarks, which is utilized in Section 4.1.1.

A.7.1 Concurrency1

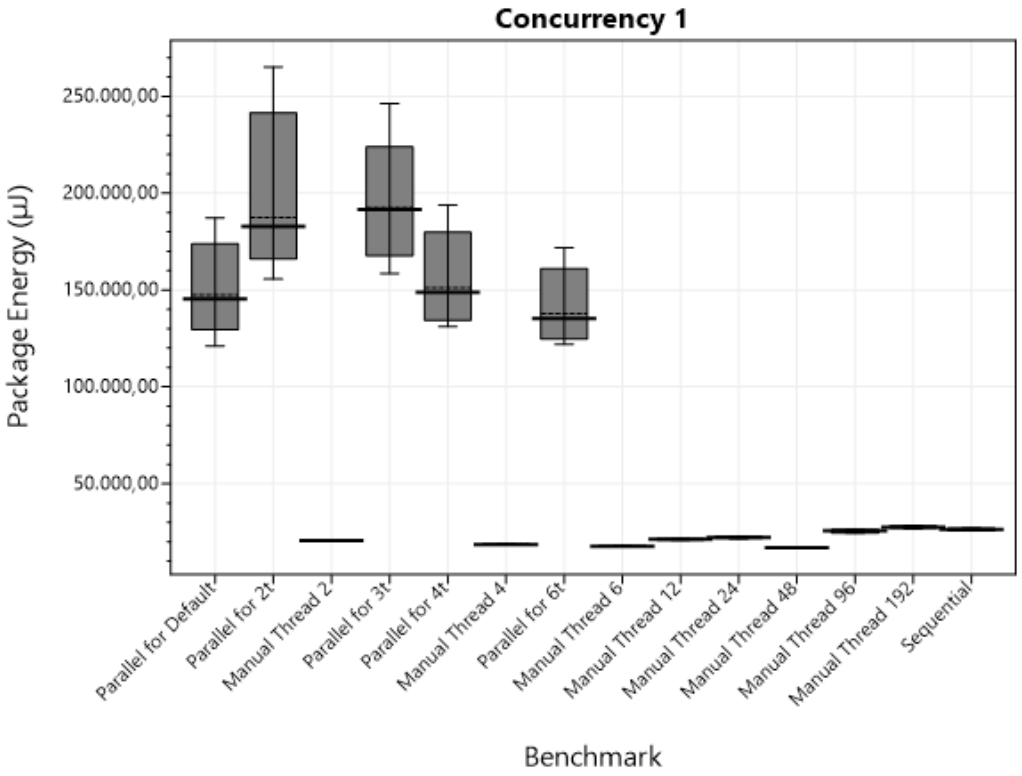

Figure A.1: Boxplot showing how efficient different Concurrency 1 types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Manual Thread 24	0,616984	22.105,548	343,233
Manual Thread 12	0,576390	21.181,991	320,748
Parallel for 2t	5,301673	187.491,304	2.969,242
Manual Thread 6	0,456451	17.626,332	253,920
Sequential	2,139497	26.316,458	1.188,608
Manual Thread 48	0,442138	17.022,944	245,988
Manual Thread 192	1,016080	27.524,666	569,109
Parallel for 4t	4,122337	151.222,146	2.322,061
Parallel for 6t	3,741768	137.710,438	2.113,499
Parallel for Default	4,081128	147.401,744	2.303,329
Manual Thread 4	0,551028	18.486,501	306,374
Manual Thread 96	0,750005	25.600,852	418,059
Manual Thread 2	1,102261	20.654,708	613,248
Using Tasks 1	445,465815	8.784.535,228	497.133,332
Parallel for 3t	5,233113	192.405,123	2.944,122
Thread Pool 1	145,510672	3.559.783,970	137.453,662

Table A.98: Table showing the elapsed time and energy measurement for each Concurrency 1.

A.7.2 Concurrency10

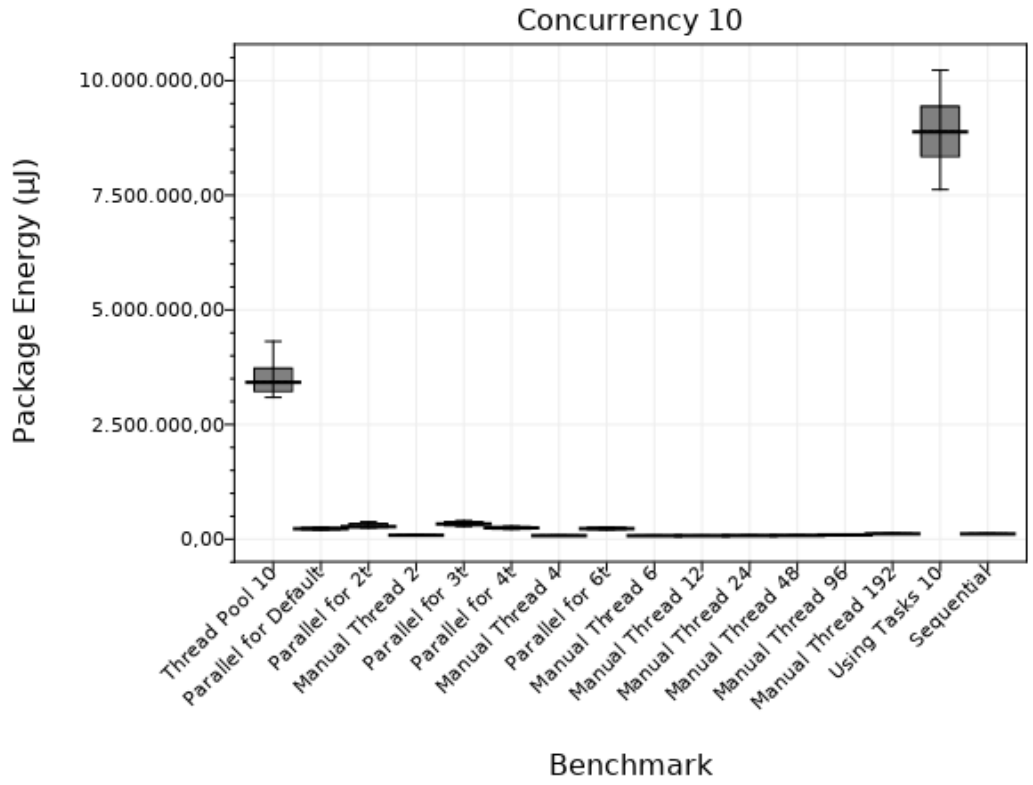

Figure A.2: Boxplot showing how efficient different Concurrency 10 types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Manual Thread 24	2,354817	76.462,575	1.310,516
Manual Thread 12	2,226941	73.794,782	1.238,875
Thread Pool 10	142,998948	3.435.874,380	140.706,831
Parallel for 2t	9,367967	279.229,081	5.235,777
Manual Thread 6	2,327642	74.114,215	1.295,345
Sequential	10,358641	114.848,694	5.761,632
Manual Thread 48	2,452644	79.254,166	1.364,845
Manual Thread 192	4,042799	116.879,626	2.264,929
Parallel for 4t	6,840168	245.199,339	3.839,335
Parallel for 6t	6,536902	230.653,654	3.682,662
Parallel for Default	6,422536	226.753,215	3.609,754
Manual Thread 4	2,632480	75.686,454	1.464,692
Using Tasks 10	463,492685	8.888.669,012	547.785,753
Manual Thread 96	2,862924	92.418,729	1.597,069
Manual Thread 2	5,266567	85.835,253	2.929,927
Parallel for 3t	9,457628	334.038,259	5.294,838

Table A.99: Table showing the elapsed time and energy measurement for each Concurrency 10.

A.7.3 Concurrency100

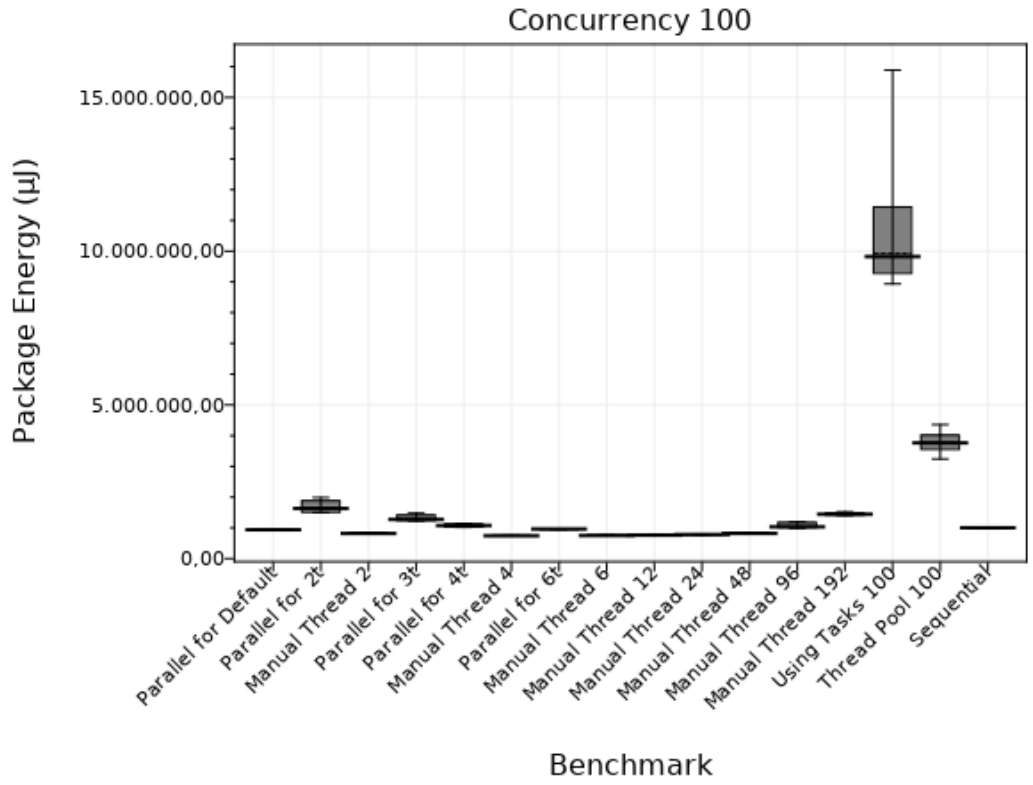

Figure A.3: Boxplot showing how efficient different Concurrency 100 types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Manual Thread 24	26,480091	784.207,861	14.734,362
Manual Thread 12	25,826973	761.800,701	14.364,639
Thread Pool 100	137,648821	3.770.541,927	126.700,641
Parallel for 2t	60,500614	1.648.720,917	33.734,273
Manual Thread 6	25,487856	751.253,852	14.170,312
Sequential	99,177122	1.003.580,451	55.164,719
Manual Thread 48	27,913941	820.535,574	15.531,854
Manual Thread 192	65,678062	1.446.983,239	36.808,699
Parallel for 4t	34,103896	1.076.241,513	19.049,427
Parallel for 6t	29,923332	962.321,872	16.719,747
Using Tasks 100	511,324079	9.928.555,290	646.087,554
Parallel for Default	29,903523	940.889,502	16.664,803
Manual Thread 4	24,688897	743.856,716	13.734,788
Manual Thread 96	37,739823	1.040.676,085	21.044,929
Manual Thread 2	49,235437	821.214,069	27.382,016
Parallel for 3t	40,958515	1.285.667,565	22.857,756

Table A.100: Table showing the elapsed time and energy measurement for each Concurrency 100.

A.7.4 Concurrency1000

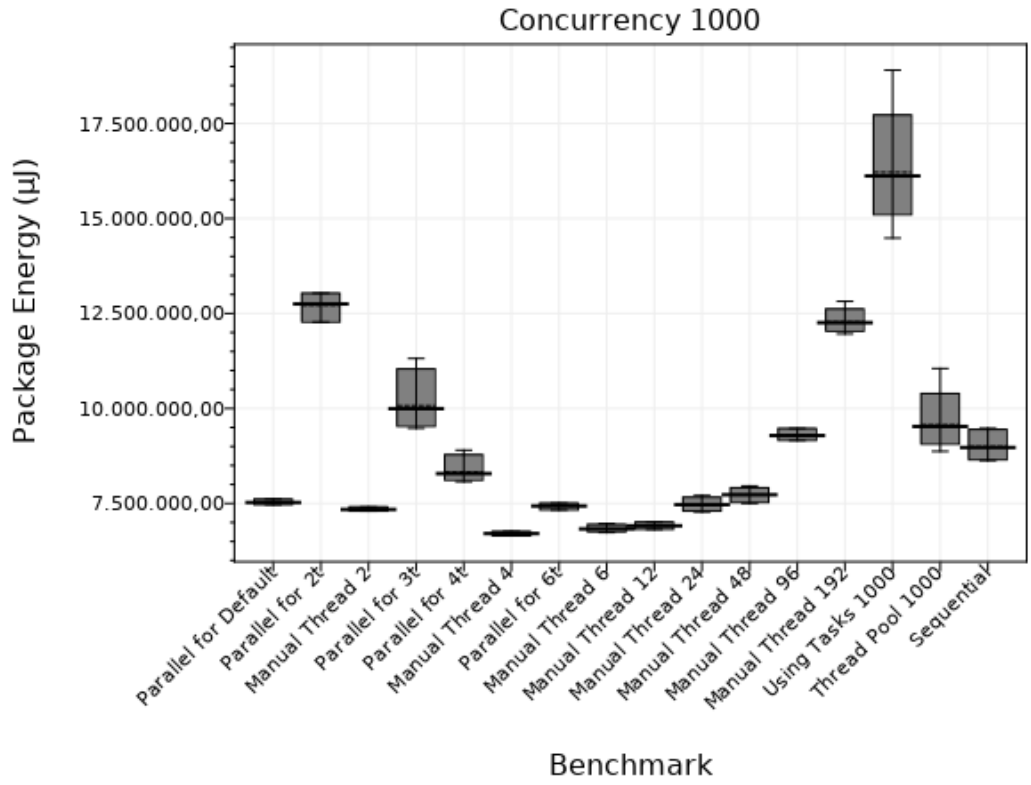

Figure A.4: Boxplot showing how efficient different Concurrency 1000 types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Manual Thread 24	255,229146	7.463.895,314	141.995,112
Manual Thread 12	231,128386	6.919.173,800	128.569,002
Using Tasks 1000	712,888916	16.216.508,760	780.287,378
Thread Pool 1000	324,738383	9.571.460,247	256.930,539
Parallel for 2t	455,439736	12.716.393,642	253.550,432
Manual Thread 6	225,408632	6.835.441,494	125.376,693
Sequential	845,346241	9.004.183,313	470.323,003
Manual Thread 48	265,135596	7.727.910,328	147.732,312
Manual Thread 192	518,875638	12.280.023,133	290.359,955
Parallel for 4t	265,606556	8.329.528,555	147.916,531
Parallel for 6t	232,513310	7.436.835,239	129.459,632
Parallel for Default	235,958395	7.526.490,307	131.514,072
Manual Thread 4	221,251516	6.703.721,237	123.047,829
Manual Thread 96	318,522814	9.292.292,867	177.433,150
Manual Thread 2	437,592745	7.339.583,111	243.493,652
Parallel for 3t	337,074322	10.077.754,486	187.705,659

Table A.101: Table showing the elapsed time and energy measurement for each Concurrency 1000.

A.7.5 Concurrency10k

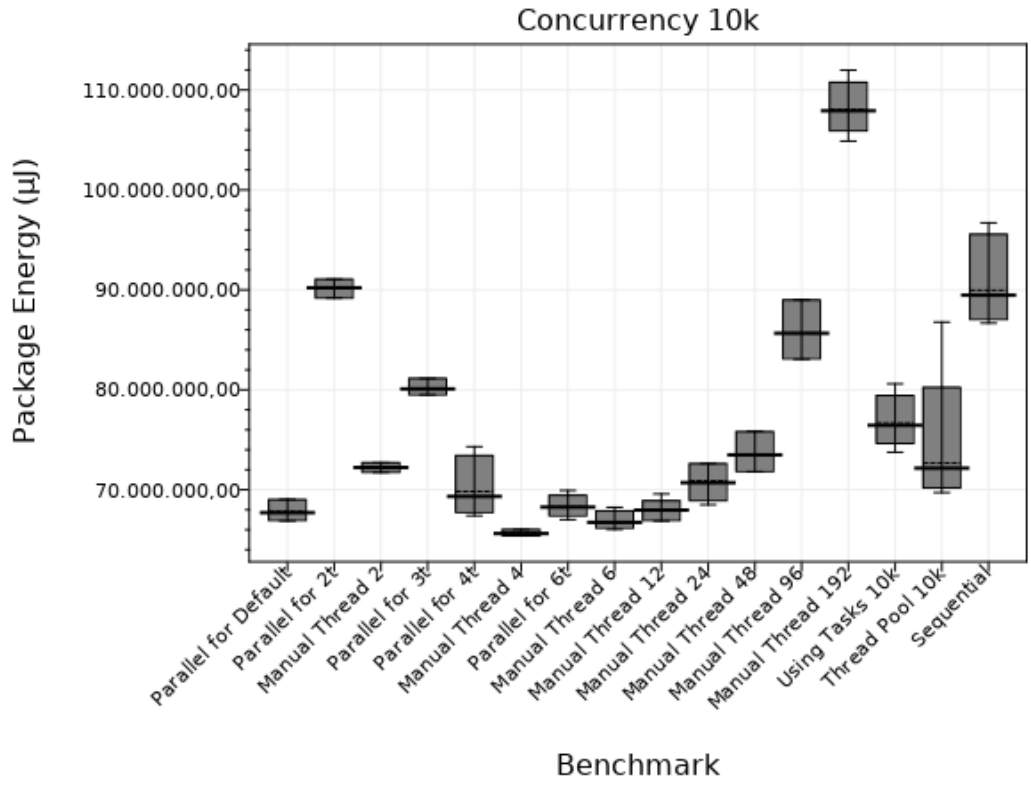

Figure A.5: Boxplot showing how efficient different Concurrency 10k types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Manual Thread 24	2.442,850770	70.888.672,876	1.360.048,013
Manual Thread 12	2.328,510666	67.903.386,116	1.295.408,076
Parallel for 2t	4.408,133589	90.004.911,150	2.452.299,935
Manual Thread 6	2.312,649519	66.776.698,373	1.286.449,375
Sequential	8.718,482826	90.018.882,025	4.848.630,633
Manual Thread 48	2.580,414779	73.593.723,156	1.437.354,618
Manual Thread 192	4.180,247251	108.053.646,980	2.342.622,992
Parallel for 4t	2.301,878505	69.836.832,682	1.280.607,276
Parallel for 6t	2.243,998680	68.297.268,518	1.248.535,881
Parallel for Default	2.229,721085	67.835.395,346	1.241.735,030
Manual Thread 4	2.202,432938	65.679.028,320	1.225.942,993
Manual Thread 96	2.981,986431	85.672.874,750	1.663.009,045
Manual Thread 2	4.348,578186	72.247.535,706	2.420.034,790
Thread Pool 10k	2.409,691411	72.670.749,919	1.402.549,388
Using Tasks 10k	2.711,358200	76.662.580,845	1.953.584,465
Parallel for 3t	2.968,949127	80.131.297,718	1.653.135,820

Table A.102: Table showing the elapsed time and energy measurement for each Concurrency 10k.

A.7.6 Concurrency100k

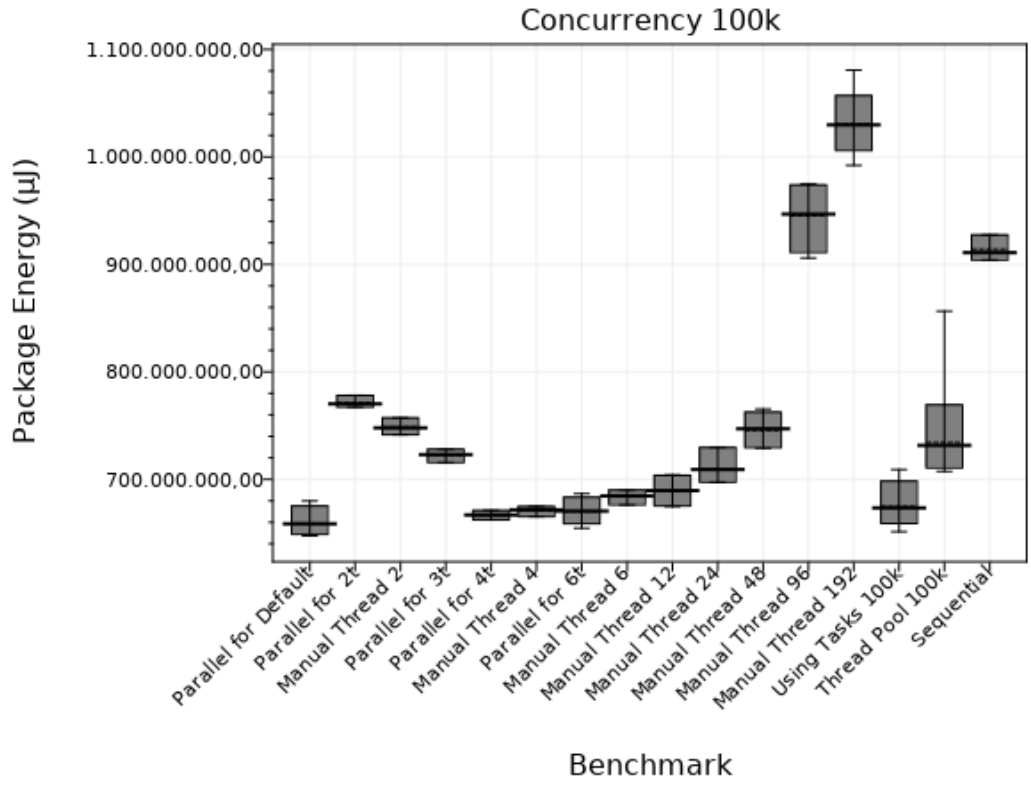

Figure A.6: Boxplot showing how efficient different Concurrency 100k types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Manual Thread 24	24.410,600891	709.278.927,612	13.588.336,182
Manual Thread 12	23.723,251272	689.833.716,104	13.197.543,832
Parallel for 2t	45.040,208740	771.649.426,270	25.040.576,172
Manual Thread 6	23.672,990723	684.218.591,309	13.170.361,328
Sequential	90.559,493001	914.507.995,605	50.358.357,747
Manual Thread 48	26.429,618779	745.551.406,295	14.709.686,279
Manual Thread 192	38.766,063544	1.030.556.548,423	21.699.205,244
Parallel for 4t	22.750,057983	666.455.328,369	12.648.071,289
Parallel for 6t	22.873,293470	669.912.354,179	12.721.860,360
Using Tasks 100k	22.961,745604	674.873.292,566	13.033.308,223
Parallel for Default	22.726,264262	659.037.703,753	12.638.495,684
Manual Thread 4	22.859,826660	670.934.985,352	12.718.109,131
Manual Thread 96	32.502,061244	945.263.654,436	18.128.677,368
Thread Pool 100k	24.344,932048	734.512.552,897	13.573.744,202
Manual Thread 2	45.240,449219	749.314.904,785	25.167.993,164
Parallel for 3t	29.978,359375	723.036.773,682	16.668.774,414

Table A.103: Table showing the elapsed time and energy measurement for each Concurrency 100k.

A.7.7 Concurrency1mil

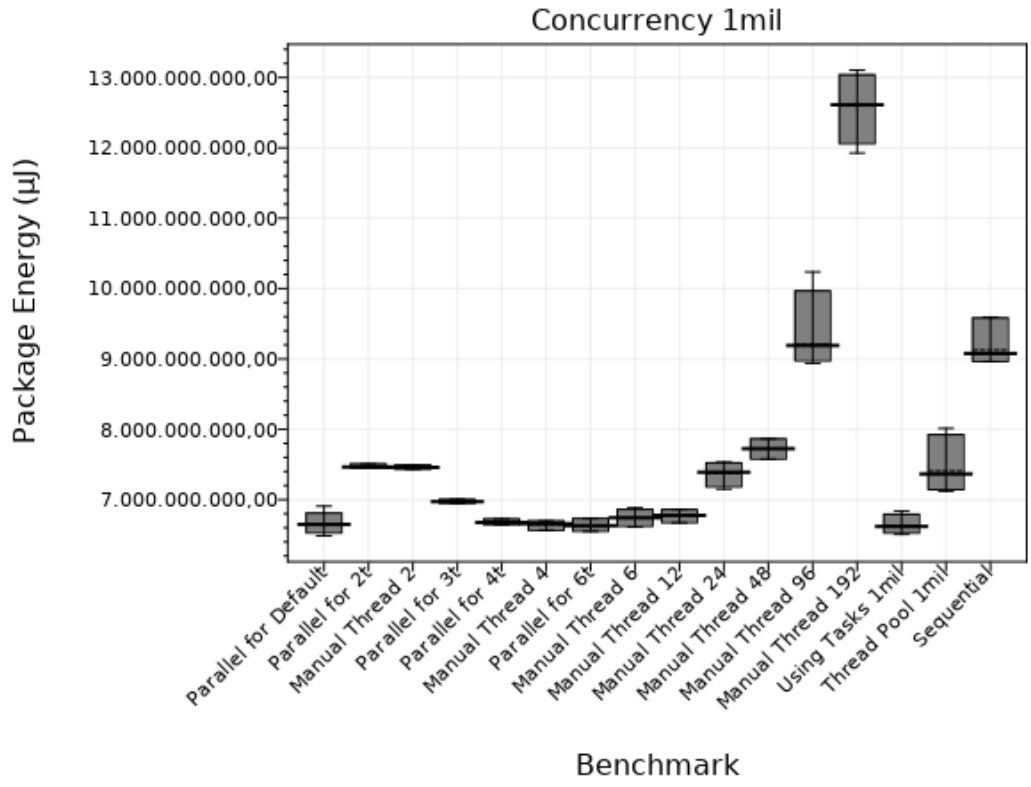

Figure A.7: Boxplot showing how efficient different Concurrency 1mil types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Manual Thread 24	263.460,636505	7.372.761.344,178	146.631.969,713
Manual Thread 12	232.768,851725	6.775.682.230,632	129.496.948,242
Parallel for 2t	449.675,400391	7.467.918.457,031	250.130.175,781
Manual Thread 6	234.368,554687	6.734.060.243,056	130.349.386,936
Using Tasks 1mil	229.179,544031	6.626.974.437,040	127.521.533,203
Sequential	899.614,382480	9.134.292.012,965	500.380.942,487
Manual Thread 48	272.810,824008	7.721.520.296,664	151.861.011,402
Manual Thread 192	563.180,711455	12.597.991.397,633	315.374.368,107
Parallel for 4t	226.887,815163	6.678.743.030,895	126.239.568,537
Parallel for 6t	227.896,879069	6.622.183.227,539	126.788.696,289
Thread Pool 1mil	245.580,109515	7.405.599.167,597	136.613.845,099
Parallel for Default	230.177,796766	6.645.743.692,236	128.057.720,907
Manual Thread 4	226.475,825195	6.645.662.500,000	126.057.519,531
Manual Thread 96	330.337,323289	9.208.785.481,771	184.074.288,504
Manual Thread 2	449.752,148438	7.458.381.640,625	250.118.359,375
Parallel for 3t	300.586,972656	6.975.358.691,406	167.256.250,000

Table A.104: Table showing the elapsed time and energy measurement for each Concurrency 1mil.

A.7.8 Concurrency10mil

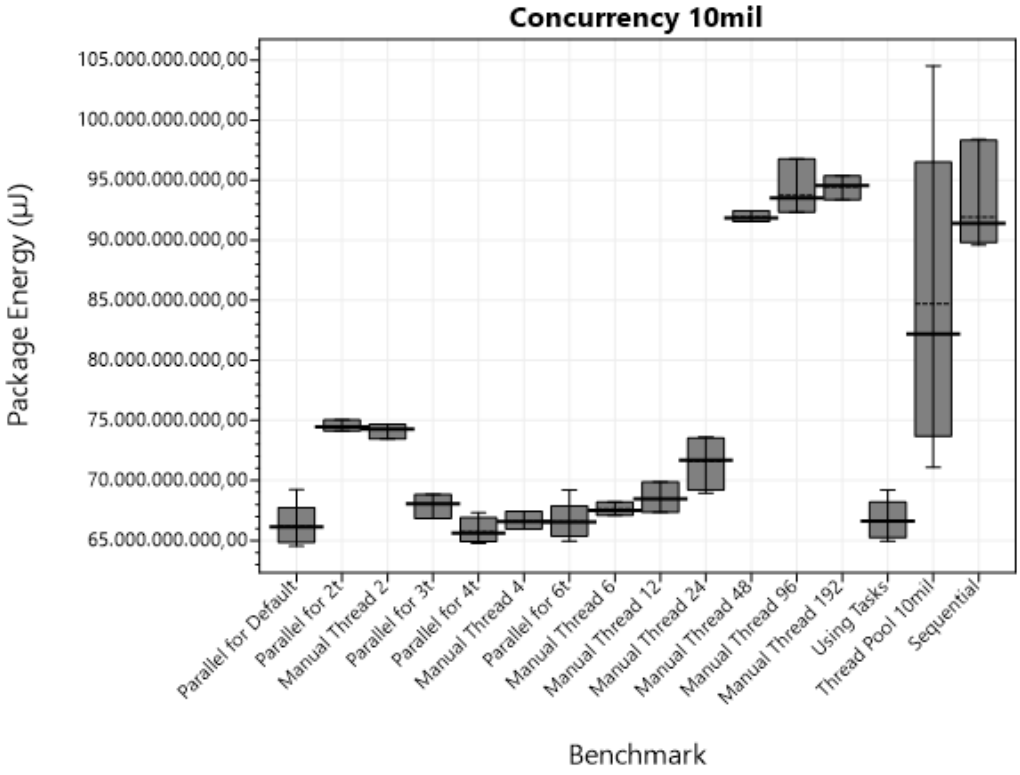

Figure A.8: Boxplot showing how efficient different Concurrency 10mil types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Manual Thread 24	2.516.085,262044	71.625.297.607,422	1.400.064.860,026
Manual Thread 12	2.397.649,915541	68.471.900.971,284	1.334.251.266,892
Parallel for 2t	4.502.227,343750	74.464.989.062,500	2.503.675.000,000
Manual Thread 6	2.384.451,523438	67.612.737.500,000	1.326.200.000,000
Sequential	9.001.406,907895	91.923.502.878,289	5.006.023.437,500
Manual Thread 48	9.012.848,750000	91.907.265.625,000	5.022.415.625,000
Manual Thread 192	9.014.904,062500	94.444.796.875,000	5.036.915.625,000
Parallel for 4t	2.305.857,836725	65.715.240.734,012	1.282.400.314,922
Thread Pool 10mil	2.772.605,809399	84.702.283.208,225	1.542.348.407,583
Parallel for 6t	2.410.649,088942	66.572.445.865,385	1.340.457.764,423
Using Tasks 10mil	2.314.745,137300	66.595.998.104,977	1.287.530.195,223
Parallel for Default	2.358.728,364583	66.172.046.205,357	1.312.102.619,048
Manual Thread 4	2.286.988,359375	66.585.275.000,000	1.271.530.468,750
Manual Thread 96	9.023.023,437500	93.733.633.522,727	5.043.796.875,000
Manual Thread 2	4.511.145,312500	74.226.570.312,500	2.508.634.375,000
Parallel for 3t	3.080.954,326923	68.003.552.884,615	1.712.023.437,500

Table A.105: Table showing the elapsed time and energy measurement for each Concurrency 10mil.

A.8 Two-sample Kolmogorov-Smirnov Test Results for Microbenchmarks

We have performed two-sample Kolmogorov-Smirnov test for all of our benchmarks by using `TwoSampleKolmogorovSmirnovTest` from the accord-framework [45]. The test are performed in regard to the package energy consumed by the different benchmarks, as this is our main area of concern. The tables presented in this section contain the p-values for these tests. The statistical significance level or α for the test is 0,05. The cell of the tables are color coded, with p-values below the significance level marked in blue, these are samples that differ significantly, and those above marked in red, which do not differ significantly. We then have from this an overview of which language construct samples differ significantly in regard to whether they are drawn from the same probability distribution.

This information is used in Section 4.1 and Section 4.2.

A.8.1 Concurrency

Concurrency 1

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	0,030	0	0,030	0,030	0	0,030	0
Parallel for 2t	0,030	-	0	0,030	0,112	0	0,309	0
Manual Thread 2	0	0	-	0	0	0,006	0	0,112
Parallel for 3t	0,030	0,030	0	-	0,309	0	0,030	0
Parallel for 4t	0,030	0,112	0	0,309	-	0	0,030	0
Manual Thread 4	0	0	0,006	0	0	-	0	0,309
Parallel for 6t	0,030	0,309	0	0,030	0,030	0	-	0
Manual Thread 6	0	0	0,112	0	0	0,309	0	-
Manual Thread 12	0	0	0,309	0	0	0,309	0	0,309
Manual Thread 24	0	0	0,664	0	0	0,030	0	0,030
Manual Thread 48	0	0	0,309	0	0	0,006	0	0,006
Manual Thread 96	0	0	0,006	0	0	0,001	0	0,006
Manual Thread 192	1,999e-5	1,999e-5	0	1,999e-5	1,999e-5	0	1,999e-5	0
Using Tasks	0	0	0	0	0	0	0	0
Sequential	0	0	0,030	0	0	0	0	1,999e-5
Thread Pool	0	0	0	0	0	0	0	0

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks	Sequential	Thread Pool
Parallel for Default	0	0	0	0	1,999e-5	0	0	0
Parallel for 2t	0	0	0	0	1,999e-5	0	0	0
Manual Thread 2	0,309	0,664	0,309	0,006	0	0	0,030	0
Parallel for 3t	0	0	0	0	1,999e-5	0	0	0
Parallel for 4t	0	0	0	0	1,999e-5	0	0	0
Manual Thread 4	0,309	0,030	0,006	0,001	0	0	0	0
Parallel for 6t	0	0	0	0	1,999e-5	0	0	0
Manual Thread 6	0,309	0,030	0,006	0,006	0	0	1,999e-5	0
Manual Thread 12	-	0,030	0,006	0,006	0	0	0,001	0
Manual Thread 24	0,030	-	0,030	0,006	0	0	0,006	0
Manual Thread 48	0,006	0,030	-	0,006	0	0	0,112	0
Manual Thread 96	0,006	0,006	0,006	-	0	0	0,112	0
Manual Thread 192	0	0	0	0	-	0	0,001	0
Using Tasks	0	0	0	0	0	-	0	0
Sequential	0,001	0,006	0,112	0,112	0,001	0	-	0
Thread Pool	0	0	0	0	0	0	0	-

Table A.106: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Concurrency 1 with regard to Package Energy.

Concurrency 10

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	0,112	0	0,309	0,112	0	0,962	0
Parallel for 2t	0,112	-	0	0,309	0,030	0	0,309	0
Manual Thread 2	0	0	-	0	0	0	0	0
Parallel for 3t	0,309	0,309	0	-	0,309	0	0,309	0
Parallel for 4t	0,112	0,030	0	0,309	-	0	0,112	0
Manual Thread 4	0	0	0	0	0	-	0	0,112
Parallel for 6t	0,962	0,309	0	0,309	0,112	0	-	0
Manual Thread 6	0	0	0	0	0	0,112	0	-
Manual Thread 12	0	0	0	0	0	0,030	0	0,112
Manual Thread 24	0	0	0,112	0	0	0,006	0	0,112
Manual Thread 48	0	0	0,112	0	0	0,030	0	0,030
Manual Thread 96	0,112	0,112	1,999e-5	0,112	0,112	0	0,112	0
Manual Thread 192	0,030	0,112	0	0,664	0,030	0	0,309	0
Using Tasks	0	0	0	0	0	0	0	0
Sequential	0,309	0,112	0	0,030	0,309	0	0,112	0
Thread Pool	0	0	0	0	0	0	0	0

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks	Sequential	Thread Pool
Parallel for Default	0	0	0	0,112	0,030	0	0,309	0
Parallel for 2t	0	0	0	0,112	0,112	0	0,112	0
Manual Thread 2	0	0,112	0,112	1,999e-5	0	0	0	0
Parallel for 3t	0	0	0	0,112	0,664	0	0,030	0
Parallel for 4t	0	0	0	0,112	0,030	0	0,309	0
Manual Thread 4	0,030	0,006	0,030	0	0	0	0	0
Parallel for 6t	0	0	0	0,112	0,309	0	0,112	0
Manual Thread 6	0,112	0,112	0,030	0	0	0	0	0
Manual Thread 12	-	0,112	0,112	0	0	0	0	0
Manual Thread 24	0,112	-	0,112	0	0	0	0	0
Manual Thread 48	0,112	0,112	-	0,006	0	0	1,999e-5	0
Manual Thread 96	0	0	0,006	-	0,006	0	0,112	0
Manual Thread 192	0	0	0	0,006	-	0	0,001	0
Using Tasks	0	0	0	0	0	-	0	0
Sequential	0	0	1,999e-5	0,112	0,001	0	-	0
Thread Pool	0	0	0	0	0	0	0	-

Table A.107: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Concurrency 10 with regard to Package Energy.

Concurrency 100

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	0,030	0,030	0,309	0,309	1,999e-5	0,962	1,999e-5
Parallel for 2t	0,030	-	0,001	0,112	0,030	0	0,030	0
Manual Thread 2	0,030	0,001	-	0,030	0,030	0	0,030	0
Parallel for 3t	0,309	0,112	0,030	-	0,309	1,999e-5	0,309	0,001
Parallel for 4t	0,309	0,030	0,030	0,309	-	0,001	0,309	0,006
Manual Thread 4	1,999e-5	0	0	1,999e-5	0,001	-	0	0,112
Parallel for 6t	0,962	0,030	0,030	0,309	0,309	0	-	0
Manual Thread 6	1,999e-5	0	0	0,001	0,006	0,112	0	-
Manual Thread 12	0,006	0	0	0,001	0,112	1,999e-5	0,006	0,112
Manual Thread 24	0,309	0	0	0,112	0,309	0	0,309	0
Manual Thread 48	0,030	0,112	0,309	0,030	0,030	0	0,030	0
Manual Thread 96	0	0,030	0	0,030	0,030	0	0	0
Manual Thread 192	0	0,030	0	0	0	0	0	0
Using Tasks	0	0	0	0	0	0	0	0
Sequential	0,001	0,030	0	0,030	0,030	0	0,001	0
Thread Pool	0	0	0	0	0	0	0	0

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks	Sequential	Thread Pool
Parallel for Default	0,006	0,309	0,030	0	0	0	0,001	0
Parallel for 2t	0	0	0,112	0,030	0,030	0	0,030	0
Manual Thread 2	0	0	0,309	0	0	0	0	0
Parallel for 3t	0,001	0,112	0,030	0,030	0	0	0,030	0
Parallel for 4t	0,112	0,309	0,030	0,030	0	0	0,030	0
Manual Thread 4	1,999e-5	0	0	0	0	0	0	0
Parallel for 6t	0,006	0,309	0,030	0	0	0	0,001	0
Manual Thread 6	0,112	0	0	0	0	0	0	0
Manual Thread 12	-	1,999e-5	0	0	0	0	0	0
Manual Thread 24	1,999e-5	-	0	0	0	0	0	0
Manual Thread 48	0	0	-	0	0	0	0	0
Manual Thread 96	0	0	0	-	0	0	0,006	0
Manual Thread 192	0	0	0	0	-	0	0	0
Using Tasks	0	0	0	0	0	-	0	0
Sequential	0	0	0	0,006	0	0	-	0
Thread Pool	0	0	0	0	0	0	0	-

Table A.108: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Concurrency 100 with regard to Package Energy.

Concurrency 1000

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	0	0,030	0,112	0,112	0,030	0,962	0,112
Parallel for 2t	0	-	0,030	0,112	0,112	0	1,999e-5	0,001
Manual Thread 2	0,030	0,030	-	0,112	0,112	0	0,030	0,006
Parallel for 3t	0,112	0,112	0,112	-	0,112	1,999e-5	0,112	0,006
Parallel for 4t	0,112	0,112	0,112	0,112	-	0,030	0,112	0,112
Manual Thread 4	0,030	0	0	1,999e-5	0,030	-	0,030	0,112
Parallel for 6t	0,962	1,999e-5	0,030	0,112	0,112	0,030	-	0,112
Manual Thread 6	0,112	0,001	0,006	0,006	0,112	0,112	0,112	-
Manual Thread 12	0,309	1,999e-5	1,999e-5	1,999e-5	0,030	0,030	0,112	0,112
Manual Thread 24	0,030	0,006	0,664	0,112	0,112	0	0,112	0,006
Manual Thread 48	0	0,112	0,030	0,112	0,112	0	1,999e-5	1,999e-5
Manual Thread 96	0	0,112	0	0,112	0	0	0	0
Manual Thread 192	0	0,112	0	0	0	0	0	0
Using Tasks	0	0	0	0	0	0	0	0
Sequential	0	0,112	0	0,112	0	0	0	0
Thread Pool	0	0,112	0	0,112	0	0	0	0

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks	Sequential	Thread Pool
Parallel for Default	0,309	0,030	0	0	0	0	0	0
Parallel for 2t	1,999e-5	0,006	0,112	0,112	0,112	0	0,112	0,112
Manual Thread 2	1,999e-5	0,664	0,030	0	0	0	0	0
Parallel for 3t	1,999e-5	0,112	0,112	0,112	0	0	0,112	0,112
Parallel for 4t	0,030	0,112	0,112	0	0	0	0	0
Manual Thread 4	0,030	0	0	0	0	0	0	0
Parallel for 6t	0,112	0,112	1,999e-5	0	0	0	0	0
Manual Thread 6	0,112	0,006	1,999e-5	0	0	0	0	0
Manual Thread 12	-	1,999e-5	1,999e-5	0	0	0	0	0
Manual Thread 24	1,999e-5	-	0,006	0	0	0	0	0
Manual Thread 48	1,999e-5	0,006	-	0	0	0	0	0
Manual Thread 96	0	0	0	-	0	0	0,030	0,006
Manual Thread 192	0	0	0	0	-	0	0	0
Using Tasks	0	0	0	0	0	-	0	0
Sequential	0	0	0	0,030	0	0	-	0,112
Thread Pool	0	0	0	0,006	0	0	0,112	-

Table A.109: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Concurrency 1000 with regard to Package Energy.

Concurrency 10k

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	0	0	0,006	0,030	0,030	0,664	0,030
Parallel for 2t	0	-	0,006	0,030	0	0,001	0	1,999e-5
Manual Thread 2	0	0,006	-	0,030	0	0,001	0	0,001
Parallel for 3t	0,006	0,030	0,030	-	0,030	0,030	0,006	0,030
Parallel for 4t	0,030	0	0	0,030	-	0,664	0,030	0,112
Manual Thread 4	0,030	0,001	0,001	0,030	0,664	-	0,112	0,030
Parallel for 6t	0,664	0	0	0,006	0,030	0,112	-	0,030
Manual Thread 6	0,030	1,999e-5	0,001	0,030	0,112	0,030	0,030	-
Manual Thread 12	0,006	0,001	0,001	0,309	0,030	0,030	0,006	0,030
Manual Thread 24	0	0,001	0,030	0,030	0	0,001	0	0,001
Manual Thread 48	0	0,309	0,006	0,030	0	0	0	0
Manual Thread 96	0	0,030	0	0	0	0	0	0
Manual Thread 192	0	0	0	0	0	0	0	0
Using Tasks	0	0,030	0,006	0,030	0	0	0	0
Sequential	0	0,030	0	0	0	0	0	0
Thread Pool	0	1,999e-5	0,112	0,030	0	0,001	0	0,001

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks	Sequential	Thread Pool
Parallel for Default	0,006	0	0	0	0	0	0	0
Parallel for 2t	0,001	0,001	0,309	0,030	0	0,030	0,030	1,999e-5
Manual Thread 2	0,001	0,030	0,006	0	0	0,006	0	0,112
Parallel for 3t	0,309	0,030	0,030	0	0	0,030	0	0,030
Parallel for 4t	0,030	0	0	0	0	0	0	0
Manual Thread 4	0,030	0,001	0	0	0	0	0	0,001
Parallel for 6t	0,006	0	0	0	0	0	0	0
Manual Thread 6	0,030	0,001	0	0	0	0	0	0,001
Manual Thread 12	-	0,001	0,001	0	0	0,001	0	0,001
Manual Thread 24	0,001	-	0,001	0	0	0,001	0	0,309
Manual Thread 48	0,001	0,001	-	0	0	0,001	0	1,999e-5
Manual Thread 96	0	0	0	-	0,001	0	0,001	0
Manual Thread 192	0	0	0	0,001	-	0	0	0
Using Tasks	0,001	0,001	0,001	0	0	-	0	1,999e-5
Sequential	0	0	0	0,001	0	0	-	0
Thread Pool	0,001	0,309	1,999e-5	0	0	1,999e-5	0	-

Table A.110: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Concurrency 10k with regard to Package Energy.

Concurrency 100k

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	0	0	0	0,309	0,030	0,112	1,999e-5
Parallel for 2t	0	-	0,309	0	0	0	0	0
Manual Thread 2	0	0,309	-	0	0	0	0	0
Parallel for 3t	0	0	0	-	0,001	0,112	0,001	0,112
Parallel for 4t	0,309	0	0	0,001	-	0,309	0,664	0,030
Manual Thread 4	0,030	0	0	0,112	0,309	-	0,309	0,309
Parallel for 6t	0,112	0	0	0,001	0,664	0,309	-	0,112
Manual Thread 6	1,999e-5	0	0	0,112	0,030	0,309	0,112	-
Manual Thread 12	0	1,999e-5	0	0,664	1,999e-5	0,112	0,001	0,309
Manual Thread 24	0	1,999e-5	1,999e-5	0,030	0	0,001	0	0
Manual Thread 48	0	0,962	0,664	0	0	0	0	0
Manual Thread 96	0	0	0	0	0	0	0	0
Manual Thread 192	0	0	0	0	0	0	0	0
Using Tasks	1,999e-5	0	0	0,664	0,006	0,664	0,006	0,309
Sequential	0	0	0	0	0	0	0	0
Thread Pool	0	0,309	0,309	0,001	0	0	0	0

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks	Sequential	Thread Pool
Parallel for Default	0	0	0	0	0	1,999e-5	0	0
Parallel for 2t	1,999e-5	1,999e-5	0,962	0	0	0	0	0,309
Manual Thread 2	0	1,999e-5	0,664	0	0	0	0	0,309
Parallel for 3t	0,664	0,030	0	0	0	0,664	0	0,001
Parallel for 4t	1,999e-5	0	0	0	0	0,006	0	0
Manual Thread 4	0,112	0,001	0	0	0	0,664	0	0
Parallel for 6t	0,001	0	0	0	0	0,006	0	0
Manual Thread 6	0,309	0	0	0	0	0,309	0	0
Manual Thread 12	-	0,001	1,999e-5	0	0	0,962	0	1,999e-5
Manual Thread 24	0,001	-	1,999e-5	0	0	0,001	0	0,001
Manual Thread 48	1,999e-5	1,999e-5	-	0	0	0	0	0,309
Manual Thread 96	0	0	0	-	0,001	0	0,030	0
Manual Thread 192	0	0	0	0,001	-	0	1,999e-5	0
Using Tasks	0,962	0,001	0	0	0	-	0	1,999e-5
Sequential	0	0	0	0,030	1,999e-5	0	-	0
Thread Pool	1,999e-5	0,001	0,309	0	0	1,999e-5	0	-

Table A.111: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Concurrency 100k with regard to Package Energy.

Concurrency 1mil

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	0	0	0	0,112	0,030	0,309	0,001
Parallel for 2t	0	-	0,309	0	0	1,999e-5	0	1,999e-5
Manual Thread 2	0	0,309	-	0	0	1,999e-5	0	0,001
Parallel for 3t	0	0	0	-	1,999e-5	0,112	1,999e-5	0,112
Parallel for 4t	0,112	0	0	1,999e-5	-	0,112	0,664	0,030
Manual Thread 4	0,030	1,999e-5	1,999e-5	0,112	0,112	-	0,112	0,309
Parallel for 6t	0,309	0	0	1,999e-5	0,664	0,112	-	0,030
Manual Thread 6	0,001	1,999e-5	0,001	0,112	0,030	0,309	0,030	-
Manual Thread 12	0	0,006	0,006	0,309	1,999e-5	0,112	1,999e-5	0,309
Manual Thread 24	0	0,030	0,112	0	0	0,001	0	0,001
Manual Thread 48	0	0	0,030	0	0	0	0	0
Manual Thread 96	0	0	0	0	0	0	0	0
Manual Thread 192	0	0	0	0	0	0	0	0
Using Tasks	0,006	0	0	0,001	0,112	0,309	0,112	0,309
Sequential	0	0	0	0	0	0	0	0
Thread Pool	0	0,006	0,006	0	0	0,001	0	0,001

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks	Sequential	Thread Pool
Parallel for Default	0	0	0	0	0	0,006	0	0
Parallel for 2t	0,006	0,030	0	0	0	0	0	0,006
Manual Thread 2	0,006	0,112	0,030	0	0	0	0	0,006
Parallel for 3t	0,309	0	0	0	0	0,001	0	0
Parallel for 4t	1,999e-5	0	0	0	0	0,112	0	0
Manual Thread 4	0,112	0,001	0	0	0	0,309	0	0,001
Parallel for 6t	1,999e-5	0	0	0	0	0,112	0	0
Manual Thread 6	0,309	0,001	0	0	0	0,309	0	0,001
Manual Thread 12	-	0,006	0,001	0	0	0,112	0	0,006
Manual Thread 24	0,006	-	0,001	0	0	0	0	0,006
Manual Thread 48	0,001	0,001	-	0	0	0	0	0,030
Manual Thread 96	0	0	0	-	0	0	0,309	0
Manual Thread 192	0	0	0	0	-	0	0	0
Using Tasks	0,112	0	0	0	0	-	0	0
Sequential	0	0	0	0,309	0	0	-	0
Thread Pool	0,006	0,006	0,030	0	0	0	0	-

Table A.112: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Concurrency 1mil with regard to Package Energy.

Concurrency 10mil

Package Energy *p*-Values	Parallel for Default	Parallel for 2t	Manual Thread 2	Parallel for 3t	Parallel for 4t	Manual Thread 4	Parallel for 6t	Manual Thread 6
Parallel for Default	-	0	0	0	0,962	0,309	0,112	0,001
Parallel for 2t	0	-	0,112	0	0	0	0	0
Manual Thread 2	0	0,112	-	0	0	0	0	0
Parallel for 3t	0	0	0	-	0	0,001	1,999e-5	0,006
Parallel for 4t	0,962	0	0	0	-	0,309	0,309	0,006
Manual Thread 4	0,309	0	0	0,001	0,309	-	0,664	0,112
Parallel for 6t	0,112	0	0	1,999e-5	0,309	0,664	-	0,006
Manual Thread 6	0,001	0	0	0,006	0,006	0,112	0,006	-
Manual Thread 12	0	0	0	0,962	0	0,001	1,999e-5	0,006
Manual Thread 24	0	0	0	0	0	0	0	0
Manual Thread 48	0	0	0	0	0	0	0	0
Manual Thread 96	0	0	0	0	0	0	0	0
Manual Thread 192	0	0	0	0	0	0	0	0
Using Tasks	0,309	0	0	0	0,664	0,112	0,112	0,006
Sequential	0	0	0	0	0	0	0	0
Thread Pool	0	1,999e-5	1,999e-5	0	0	0	0	0

Package Energy *p*-Values	Manual Thread 12	Manual Thread 24	Manual Thread 48	Manual Thread 96	Manual Thread 192	Using Tasks	Sequential	Thread Pool
Parallel for Default	0	0	0	0	0	0,309	0	0
Parallel for 2t	0	0	0	0	0	0	0	1,999e-5
Manual Thread 2	0	0	0	0	0	0	0	1,999e-5
Parallel for 3t	0,962	0	0	0	0	0	0	0
Parallel for 4t	0	0	0	0	0	0,664	0	0
Manual Thread 4	0,001	0	0	0	0	0,112	0	0
Parallel for 6t	1,999e-5	0	0	0	0	0,112	0	0
Manual Thread 6	0,006	0	0	0	0	0,006	0	0
Manual Thread 12	-	0	0	0	0	0	0	0
Manual Thread 24	0	-	0	0	0	0	0	0
Manual Thread 48	0	0	-	0,309	0,001	0	0,006	0
Manual Thread 96	0	0	0,309	-	0,112	0	0,006	0
Manual Thread 192	0	0	0,001	0,112	-	0	0,001	0
Using Tasks	0	0	0	0	0	-	0	0
Sequential	0	0	0,006	0,006	0,001	0	-	0
Thread Pool	0	0	0	0	0	0	0	-

Table A.113: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Concurrency 10mil with regard to Package Energy.

A.8.2 Casting

Numeric Casting

Package Energy *p*-Values	Implicit	Explicit
Implicit	-	0,309
Explicit	0,309	-

Table A.114: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Numeric Casting with regard to Package Energy.

Derived Casting

Package Energy *p*-Values	Reference Explicit	As
Reference Explicit	-	0,112
As	0,112	-

Table A.115: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Derived Casting with regard to Package Energy.

A.8.3 Parameter Mechanisms

Non Modifying

Package Energy *p*-Values	In	Ref
In	-	0,664
Ref	0,664	-

Table A.116: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Non modifying use of parameter mechanisms with regard to Package Energy.

Modifying reference-type

Package Energy *p*-Values	Ref Mod	Out
Ref Mod	-	0,962
Out	0,962	-

Table A.117: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of modifying reference types with use of parameter mechanisms, with regard to Package Energy.

Modifying value-type

Package Energy *p*-Values	Return Large Struct	Ref Large Struct	Return Small Struct	Ref Small Struct
Return Large Struct	-	0	0	0
Ref Large Struct	0	-	0	0,962
Return Small Struct	0	0	-	0
Ref Small Struct	0	0,962	0	-

Table A.118: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of modifying value types, in the form of structs, with use of parameter mechanisms, with regard to Package Energy.

Returning

Package Energy *p*-Values	Ref Return	Out Return	Return	Return Alt
Ref Return	-	0,962	0,664	0,664
Out Return	0,962	-	0,664	0,664
Return	0,664	0,664	-	0,664
Return Alt	0,664	0,664	0,664	-

Table A.119: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of using parameter mechanisms to return values with regard to Package Energy.

A.8.4 Lambda

Package Energy *p*-Values	Lambda	Lambda Closure	Lambda Param	Lambda Delegate	Lambda Delegate Closure	Lambda Action
Lambda	-	0,030	0,308	0,030	0,006	0,030
Lambda Closure	0,030	-	0,112	0,112	0,001	0,309
Lambda Param	0,308	0,112	-	0,112	0,001	0,112
Lambda Delegate	0,030	0,112	0,112	-	0,001	0,006
Lambda Delegate Closure	0,006	0,001	0,001	0,001	-	0,001
Lambda Action	0,030	0,309	0,112	0,006	0,001	-

Table A.120: Table showing the *p*-values for the two-sample Kolmogorov-Smirnov test of Lambda Expressions with regard to Package Energy.

A.9 Two-sample Kolmogorov-Smirnov Test Results for Macrobenchmarks

We have also performed two-sample Kolmogorov-Smirnov test for our macrobenchmarks. The test are performed in regard to the package energy con-

sumed, we here present the p-values for these tests. The statistical significance level or α for the test is 0,05. The cell of the tables are color coded, with p-values below significance level marked in blue and those above marked in red.

This information is used in Section 5.2 and Section 5.3.

A.9.1 100 Prisoners

Package Energy p-Values	Original	Parallel for	No LINQ	Array	Array No LINQ	Parallel for No LINQ	Parallel for Array	Parallel for Array No LINQ
Original	-	0	0	1,999E-05	0	0	0	0
Parallel for	0	-	0	0	0	0	0,112	0
No LINQ	0	0	-	0	0	0	0	0
Array	1,999E-05	0	0	-	0	0	0	0
Array No LINQ	0	0	0	0	-	0	0	0
Parallel for No LINQ	0	0	0	0	0	-	0	0,006
Parallel for Array	0	0,112	0	0	0	0	-	0
Parallel for Array No LINQ	0	0	0	0	0	0,006	0	-

Table A.121: Table showing the p-values for the two-sample Kolmogorov-Smirnov test of 100 Prisoners with regard to Package Energy.

A.9.2 Almost Prime

Package Energy p-Values	Original	Manual Thread	Array	Struct	Manual Thread Array	Manual Thread Struct	Array Struct	Manual Thread Array Struct
Original	-	0	0,112	0,309	0	0	0,664	0
Manual Thread	0	-	0	0	0,664	0,112	0	0,664
Array	0,112	0	-	0,112	0	0	0,030	0
Struct	0,309	0	0,112	-	0	0	0,309	0
Manual Thread Array	0	0,664	0	0	-	0,962	0	0,962
Manual Thread Struct	0	0,112	0	0	0,962	-	0	0,664
Array Struct	0,664	0	0,030	0,309	0	0	-	0
Manual Thread Array Struct	0	0,664	0	0	0,962	0,664	0	-

Table A.122: p-values for the two-sample Kolmogorov-Smirnov test of Almost Prime with regard to Package Energy.

A.9.3 Chebyshev Coeffecients

Package Energy p-Values	Original	Concurrent	Array	Datatype	Concurrent Array	Concurrent Datatype	Array Datatype	Concurrent Array Datatype
Original	-	0	0	0,309	0	0	0,006	0
Concurrent	0	-	0	0	0	0,309	0	0
Array	0	0	-	0	0	0	0,001	0
Datatype	0,309	0	0	-	0	0	0	0
Concurrent Array	0	0	0	0	-	0	0	0,962
Concurrent Datatype	0	0,309	0	0	0	-	0	0
Array Datatype	0,006	0	0,001	0	0	0	-	0
Concurrent Array Datatype	0	0	0	0	0,962	0	0	-

Table A.123: p-values for the two-sample Kolmogorov-Smirnov test of Chebyshev Coeffecients with regard to Package Energy.

A.9.4 Compare String Length

Package Energy *p*-Values	Original	Selection	Parallel for 1	Parallel for 2	Foreach	Selection Foreach
Original	-	0,664	0	0	0	0
Selection	0,664	-	0	0	0	0
Parallel for 1	0	0	-	0	0	0
Parallel for 2	0	0	0	-	0	0
Foreach	0	0	0	0	-	0,962
Selection Foreach	0	0	0	0	0,962	-
Selection Parallel for 1	0	0	0,962	0	0	0
Selection Parallel for 2	0	0	0	0,309	0	0
Parallel for 1 Foreach	0	0	0	0	0	0
Parallel for 2 Foreach	0	0	0	0,030	0	0
Selection Parallel for 1 Foreach	0	0	0	0	0	0
Selection Parallel for 2 Foreach	0	0	0	0,006	0	0

Package Energy *p*-Values	Selection Parallel for 1	Selection Parallel for 2	Parallel for 1 Foreach	Parallel for 2 Foreach	Selection Parallel for 1 Foreach	Selection Parallel for 2 Foreach
Original	0	0	0	0	0	0
Selection	0	0	0	0	0	0
Parallel for 1	0,962	0	0	0	0	0
Parallel for 2	0	0,309	0	0,030	0	0,006
Foreach	0	0	0	0	0	0
Selection Foreach	0	0	0	0	0	0
Selection Parallel for 1	-	0	0	0	0	0
Selection Parallel for 2	0	-	0	0,112	0	0,112
Parallel for 1 Foreach	0	0	-	0	0,664	0
Parallel for 2 Foreach	0	0,112	0	-	0	0,962
Selection Parallel for 1 Foreach	0	0	0,664	0	-	0
Selection Parallel for 2 Foreach	0	0,112	0	0,962	0	-

Table A.124: *p*-values for the two-sample Kolmogorov-Smirnov test of Compare String Length with regard to Package Energy.

A.9.5 Dijkstra's Algorithm

Package Energy *p*-Values	Original	No Linq	Array	Objects	No Linq Array	No Linq Objects	Array Objects	No Linq Array Objects
Original	-	0	0,309	0,112	0	0	0,030	0
No Linq	0	-	0	0	0,309	0,112	0	1,999E-05
Array	0,309	0	-	0,309	0	0	0,664	0
Objects	0,112	0	0,309	-	0	0	0,962	0
No Linq Array	0	0,309	0	0	-	0,006	0	0,030
No Linq Objects	0	0,112	0	0	0,006	-	0	0
Array Objects	0,030	0	0,664	0,962	0	0	-	0
No Linq Array Objects	0	1,999E-05	0	0	0,030	0	0	-

Table A.125: *p*-values for the two-sample Kolmogorov-Smirnov test of Dijkstra's Algorithm with regard to Package Energy.

A.9.6 Four Squares Puzzle

Package Energy *p*-Values	Original	Parallel for	No LINQ	Selection	Parallel for No LINQ	Parallel for Selection	No LINQ Selection	Parallel for No LINQ Selection
Original	-	0	0	1,999E-05	0	0	0	1,999E-05
Parallel for	0	-	0	0	0	0,112	0	0
No LINQ	0	0	-	0	0	0	0,962	0
Selection	1,999E-05	0	0	-	0	0	0	0
Parallel for No LINQ	0	0	0	0	-	0	0	0,112
Parallel for Selection	0	0,112	0	0	0	-	0	0
No LINQ Selection	0	0	0,962	0	0	0	-	0
Parallel for No LINQ Selection	1,999E-05	0	0	0	0,112	0	0	-

Table A.126: *p*-values for the two-sample Kolmogorov-Smirnov test of Four Squares Puzzle with regard to Package Energy.

A.9.7 Numbrix Puzzle

Package Energy *p*-Values	Original	Linq	Parallel for	Selection	Linq Parallel for	Linq Selection	Parallel for Selection	Linq Parallel for Selection
Original	-	1,999E-05	0	0,664	0	0	0	0
Linq	1,999E-05	-	0	0,001	0	0,309	0	0
Parallel for	0	0	-	0	0,962	0	0,962	0,664
Selection	0,664	0,001	0	-	0	0	0	0
Linq Parallel for	0	0	0,962	0	-	0	0,962	0,664
Linq Selection	0	0,309	0	0	0	-	0	0
Parallel for Selection	0	0	0,962	0	0,962	0	-	0,962
Linq Parallel for Selection	0	0	0,664	0	0,664	0	0,962	-

Table A.127: *p*-values for the two-sample Kolmogorov-Smirnov test of Numbrix Puzzle with regard to Package Energy.

A.9.8 Sum to Hundred

Package Energy *p*-Values	Original	No Linq Concurrent	Selection	Objects
Original	-	0	0,664	0
No Linq Concurrent	0	-	0	0
Selection	0,664	0	-	0
Objects	0	0	0	-
No Linq Concurrent Selection	0	0,664	0	1,999E-05
No Linq Concurrent Objects	0	0	0	0
Selection Objects	0	0	0	0,112
No Linq Concurrent Selection Objects	0	0	0	0

Package Energy *p*-Values	No Linq Concurrent Selection	No Linq Concurrent Objects	Selection Objects	No Linq Concurrent Selection Objects
Original	0	0	0	0
No Linq Concurrent	0,664	0	0	0
Selection	0	0	0	0
Objects	1,999E-05	0	0,112	0
No Linq Concurrent Selection	-	0	1,999E-05	0
No Linq Concurrent Objects	0	-	0	0
Selection Objects	1,999E-05	0	-	0
No Linq Concurrent Selection Objects	0	0	0	-

Table A.128: *p*-values for the two-sample Kolmogorov-Smirnov test of Sum to Hundred with regard to Package Energy.

A.10 Macrobenchmark Changes

In Section 5.1 we present the need for a list of changes we make to benchmarks, this section contains this list. Each section contains the different changes we can make to a particular group of language constructs on the benchmark. The benchmark are evaluated to determine which of these groups can be changed in them and what parts within the group can be changed.

A.10.1 Concurrency

Concurrency at low workloads should be done using manual thread control, fitted to the amount of available cores.

When the workload grows substantially large enough `Task` and `Parallel.For` are comparable choices to using manual threading and are as such interchangeable when concerned with energy consumption.

A.10.2 Parameter Mechanisms

Modify methods that use pass-by-value and `return` in conjunction with structure types, to pass by reference by using `ref`.

A.10.3 Lambda

When using lambda expressions avoid using delegates when able to do so. Use of closure should also be avoided due to the capture of outer variables and variable scope [36] imposing higher cost in the form of garbage collection.

A.10.4 Datatypes

Integer variables should use `uint` as the first option, then followed by `ulong`. In cases where unsigned integers is not a possibility the order of preferable integer datatype is: `int`, `nint` and long.

For decimal variables the preferable option is using double.

When looking at type of variable used parameters are preferable, if these cannot be used then local variables are preferable to instance and static variables, with static variables being preferred to instance variables.

A.10.5 Selection

Replace `if` statements with `switch` statements when applicable.

A.10.6 Loops

When iterating through elements in a collection, the use of `foreach` is preferable to `for`.

Language Integrated Query (LINQ) should be avoided whenever possible and should be substituted by `for` or `foreach`.

A.10.7 Collections

Replace use of any type of list with an `array` if possible. If this is not possible, the best type of list to use is `List<T>`.

Replace use of any type of set with `HashSet<T>`, when able to do so.

Using `Dictionary<TKey,TValue>` is preferable to any other type of table, so it should be used when the situation allows for it.

A.10.8 Objects

Use `structs` instead of `classes` and `records`, when invocation and creation of `methods` and `objects` is a common operation.