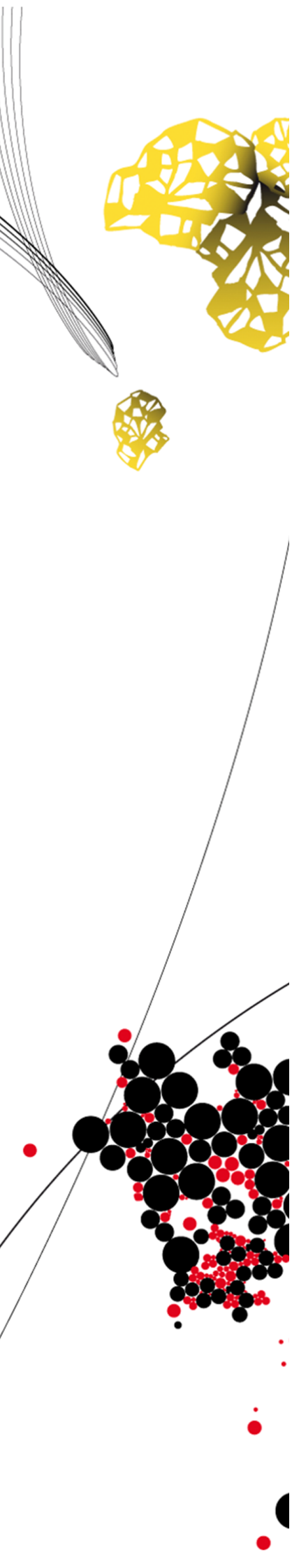

UNIVERSITY OF TWENTE.

Formal Methods & Tools

Impacts of programming environments and practices on energy consumption

Tycho L. Braams

M.Sc. Thesis
August 2020

Supervisors:
dr. L. Ferreira Pires
dr. T. van Dijk
dr. A. Fehkner
H. Logmans (Alten)

Formal Methods & Tools
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Energy consumption and sustainability are of increasing importance in our current society. In the early 2000s, predictions on the growth of energy consumption of ICT were worrying. Due to improvements made in hardware development to reduce idle consumption, the reduction of overhead costs in data centres and the use of more efficient devices, the predicted growth was not reached. Since the use of ICT and the amount of data being transferred continues to increase, it is important to look at other possibilities for reducing energy consumption.

We looked at the energy consumption of Object-Oriented software, specifically focusing on C#. By performing empirical experiments, we developed a methodology for performing energy consumption measurements and we analysed the impact of compiler settings on the energy consumption of software. We found that the compiler settings can have a different impact based on the software being run, but there was one setting that performed the worst for both energy consumption and execution time. This setting should be avoided, while further analysis is necessary to discover if the differing impact can be linked to programming structures. We also propose experiments for analysing the energy consumption of programming structures which could lead to guidelines for programmers.

Contents

List of acronyms

ICT	Information & Communication Technology
RAPL	Running Average Power Limit
MSR	Machine Specific Registers
CPU	Central Processing Unit
JIT	Just-in-time
Q	Quick JIT
L	Quick JIT for loops
R	ReadyToRun
T	Tiered Compilation
SMM	System Management Mode
SFL	Spectrum-based Fault Localization
SPELL	Spectrum-based Energy Energy Leak Localization
CLBG	Computer Language Benchmark Game
GHC	Glassgow Haskell Compiler
VB	Visual Basic
LINQ	Language Integrated Query

Chapter 1

Introduction

1.1 Motivation

Currently, climate change, resource usage and energy use are important points of discussion. It has been suggested that Information & Communication Technology (ICT) solutions can be used to reduce the energy use of other industries or to improve processes to reduce the emission of greenhouse gases [1]. Such efforts are often referred to as Greening by ICT. It is also important to look at the energy consumption of the ICT sector itself. In the early 2000's multiple research projects made estimations about energy consumption and made predictions about future trends based on observed trends. These estimations indicated that the emissions caused by the production of the energy consumed by the ICT sector were equal to 2% of global emissions [2]. Furthermore, the predictions on future energy consumption showed a strong growth that would likely become untenable [3]. Later research found that this growth had slowed down, partly due to improvements made by hardware producers, which were sometimes induced by governmental policies such as Energy Star [4]. New technologies such as mobile devices and increased use of laptops over desktops also contributed, since these devices are inherently more efficient [5]. Furthermore, these devices also provide an incentive to reduce energy consumption since the energy available on the devices is limited by their battery capacity. Data centre operators also worked on reducing their energy consumption in order to reduce operating costs by improving the effectiveness of cooling and increasing the use of virtualisation to make better use of the available resources [6]. Recently, it was estimated that these reduction efforts have caused the energy consumption of data centres to plateau instead of continuing to grow.

Research also created and analysed methods to measure the energy consumption of specific machines or programs [7]. In 2011, Intel introduced the Running Average Power Limit (RAPL) which can be used to manage the power usage of the Central Processing Unit (CPU) [8]. It also introduced registers that keep track of the

energy consumed by the CPU. Research has shown that this is very accurate [9]. Different measurement methods have been used to analyse the energy consumption of software. They have been used to find inefficiencies in energy consumption of mobile devices [10], rank programming languages according to energy consumption [11] [12], analyse the energy consumption of different data structure implementations in Java [13] and Haskell [14], analyse the impact of Design patterns [15], etc. These efforts have shown that it is possible to obtain significant reductions in the energy consumed by software.

1.2 Project background

An initial literature study was performed on the energy consumption of ICT. This showed that it is a new field of study. It also became clear that programmers sometimes have vague ideas about what could reduce the energy consumption of their software, they lacked evidence to support these ideas. We decided to look into the energy consumption of Object-Oriented programming. Some research had already been done on the energy consumption of Java, partially since it is linked with Android. The energy consumption of application on mobile systems is particularly important as they have a limited battery capacity. As the literature study also showed that energy reduction efforts in data centres had already reduced most of the overhead, we decided to focus on C#. C# is another popular Object-Oriented programming languages, often used for web-development and back-end systems. If it is possible to reduce the energy consumption of C# software, this could lead to a reduction in the energy consumption of software running in data centres. We decided to focus our research on the compiler options provided by the C# development framework.

1.3 Goals

As mentioned previously, there has been a lot of research on the energy consumption of ICT and software in particular. However, it is not yet clear to practitioners how they can manage energy consumption in their projects. Research is often focused on a single platform and programming language, with researchers pointing out that it is unknown if or how their research can be generalised. In this research, we hope to make a step towards bridging the gap between researchers and practitioners. The first step is to find out if it is feasible for practitioners to make energy consumption measurements so that they can become aware of the energy consumption of the software they are creating. The next step is to help practitioners make choices to

reduce the energy consumption of their software.

Specifically, we focus on the following research question: **RQ**: How can the energy consumption of software systems be reduced? To answer this question, several sub-questions have been defined that split the problem into smaller steps.

RQ1: How can developers obtain reliable results on the energy consumption of their software? If developers want to make use of energy consumption information, it should be clear how they can obtain reliable results. Complicated initialisation steps or restrictive settings make it less likely that the average developer will perform such measurements. It should be investigated how performing measurements can be made accessible while still producing reliable results.

RQ2: How do hardware settings influence the energy consumption of software systems? There are many different hardware settings that can be changed. It is useful to know how such settings impact energy consumption measurements.

RQ3: How do C# compiler settings influence the energy consumption of software systems? .NET Core offers several compiler settings. These settings were introduced to reduce the start-up time associated with the Just-in-time (JIT) compiler. It is not yet clear if and how these settings impact the energy consumption of software.

RQ4: How consistent is the impact of compiler settings across hardware architectures? In order to make decisions about which compiler setting to use, it is important to know if the impact of such settings is consistent across different architectures.

RQ5: How do functionally equal programming choices impact energy consumption? In order to reduce the energy consumption of software, it is necessary to investigate if different choices can have an impact on energy consumption.

To answer research questions 1 to 4, empirical measurements are performed using existing benchmarks. We have no empirical measurements to answer research question 5, but we provide code examples that could show energy consumption differences.

1.4 Contributions

This thesis adds to the body of knowledge on energy consumption of software. In particular, hardware measurement showed results similar to those reported in a paper published during the course of this research. Furthermore, the influence of compiler options offered in .NET Core on energy consumption is analysed. No research was found that treated the influence of these compiler options. We also expanded on previous research to create a methodology for performing energy consumption

measurements that can be used in future research. Finally, a step is made towards making the results of research usable by practitioners.

1.5 Overview

The rest of the report is structured as follows. In Chapter 2, related work is discussed. We discuss research on how the energy consumption of software can be measured and research on the impact of programming choices on the energy consumption software. In Chapter 3, the methodology used in the research is described. In Chapter 5, the initial measurements on energy consumption are analysed. In Chapter 6, the impact of hardware settings on energy consumption is discussed. In Chapter 7, the impact of compiler settings on energy consumption is discussed. In Chapter 8, the results of performing the experiments on different hardware is discussed. In Chapter 9, we make suggestions for experiments that could be performed to measure the energy consumption of programming choices. Finally, in Chapter 10, we summarise validity threads, conclusions of the research and possibilities for future research.

Chapter 2

Related Work

In this chapter, we discuss related research on energy consumption. In particular, we look at methods to measure energy consumption of IT devices in Section 2.1 and how such methods have been used to analyse the energy consumption of software in Section 2.2.

2.1 Energy measurement

Ghaleb [7] analysed different methods for measuring the power and energy consumption of software programs. Based on this analysis, a taxonomy is proposed to classify these methods into multiple categories. Hardware methods make use of specialised devices that contain sensors to perform measurements or use a power meter to measure the usage via the power supply. Software methods make use of models or system variables to estimate the energy consumption of the device. They focused on looking at the sampling frequency, measurement granularity and the hardware components that are measured. Most software methods have a lower sample frequency, making it more difficult to obtain energy consumption estimations/measurements for lower levels of software or specific sections of code. Hardware methods have a higher sample frequency but offer less information on which component is consuming the energy, mostly limited on information for the entire machine. Ghaleb et al. did not look at the accuracy of measurement results, only looking at which methods are available. Jagroep et al. [16] analysed different software tools that can be used to measure energy consumption. They selected fourteen energy profilers but were only able to successfully install six of them. Furthermore, of these six, they only managed to get two energy profilers fully operational. They encounter problems with configuring the other profilers, where even contacting the developers of the tools did not help them in fixing the issues they encountered. They performed measurements with the two profilers they managed to get operational.

They found significant differences when comparing the profilers estimations to measurements obtained from hardware tools. To improve the results of the profilers, they calculated correction factors for both tools, that should be applied to their results. However, they did find that the profilers were timely with their measurements, and could be used to get a feel for the trends in energy consumption.

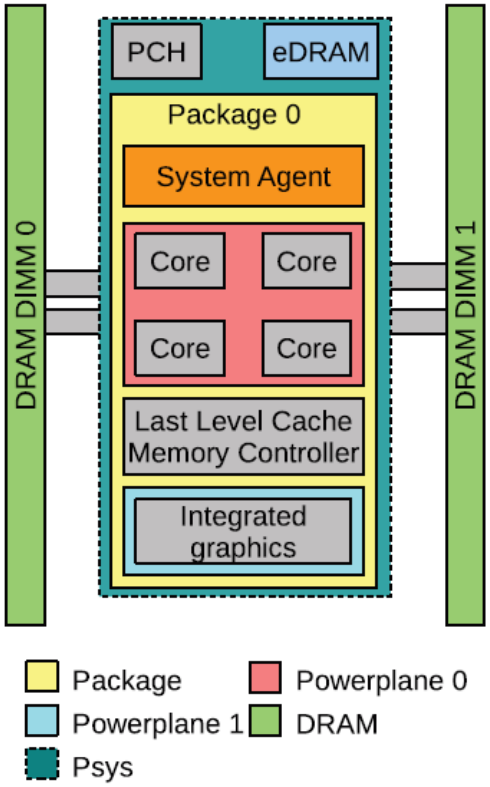

Figure 2.1: Power domains supported by RAPL, by Khan et al. [9]

In 2011 Intel introduced the RAPL interface with the Sandy Bridge micro-architecture. This interface can be used to manage the power usage and temperature of the CPU. It also provides Machine Specific Registers (MSR) that contain information about the energy consumption of different "domains". Figure 2.1 displays the different domains supported by RAPL. Domain support varies for different processor models. The Package domain is universally supported. The Psys domain was introduced with the Skylake architecture but requires extra system-level implementations and is therefore not supported in all Skylake versions.

Hähnel et al. [17] analysed the suitability of using RAPL to perform energy measurements on short code paths. They looked at the update frequency of the machine registers by continually reading the registers and a time stamp. They found that most register updates occur within a range of 2% above and below the expected update time. A small number of updates showed a significant delay They deduced that these delays are caused by the CPU switching into System Management Mode (SMM),

which cannot be controlled by the Operating system. By experimenting, they found that this occurs every 16 ms. Based on these findings, they created a framework to measure the energy consumption of short code paths. This framework consists of multiple steps. First, a loop delays the execution of the code under test until a RAPL update is detected. Then the code is executed and once it is finished, another loop reads the register to detect the next update. They found that reading the register introduced a constant energy cost, so by counting the number of register reading operations, they can subtract the cost of this loop from the total energy consumption to find the consumption of the code under test. They also note that it is possible to delay execution until a delay caused by the CPU entering SMM is detected. This does limit the possible execution time of the code to under 16 ms to miss the next SMM delay. Finally, Hähnel et al. compared the results of RAPL energy consumption measurements to external measurements. They found that there was a consistent offset between the two measurements. This can be explained by the fact that RAPL measurements are limited to the CPU, while the external measurement includes other components that also consume energy.

Spencer et al. [18] added to the validation of RAPL measurements by analysing the performance of DRAM measurements. They analysed multiple types of memory using multiple tests. They measured the consumption under load by the CPU, idle and under load by the GPU. They found that the behaviour differed based on the type of memory. The RAPL measurements differed up to 20%. However, the differences were constant, meaning that the measurements accurately tracked the behaviour of the energy consumption, with an offset compared to actual measurements. The largest differences were encountered when the system was idle or when the memory is being used by the GPU. Finally, they found that Haswell-EP server machines use actual measurements, while earlier architectures provided estimations. This improves the accuracy of the energy consumption reported byRAPL.

Khan et al. [9] studied several aspects of RAPL such as accuracy and granularity. They studied the results reported by different architectures. They found that the introduction of on-chip voltage regulators in Haswell considerably improved the results compared to Sandybridge. In the Skylake architecture, the PP0 domain is updated every 50-70 μs, a considerable change compared to Haswell with updates approximately every 1 ms. This improves possibilities to measure the energy consumption of short code paths. Khan et al. also investigated if it is possible to use RAPL to identify different execution phases of a program. The results of one such test can be seen in Figure 2.2. This shows the measurements provided by RAPL as well as the results by measuring the power consumption of the plug at the wall socket. A test with a different benchmark showed the impact of different sampling rates. The wall power was measured with a sampling rate of 100ms while RAPL was

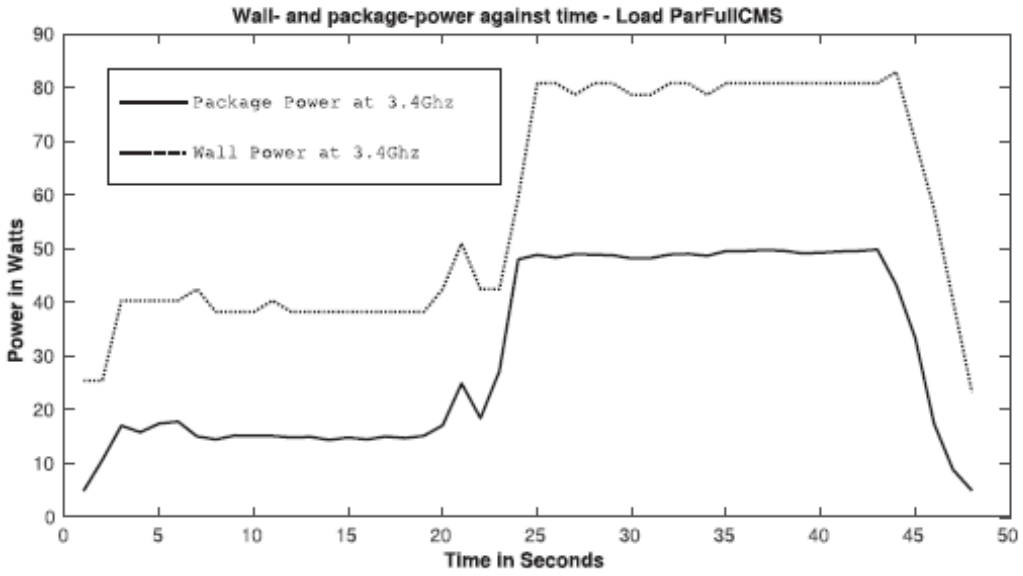

Figure 2.2: Wall and RAPL package power consumption with time.

sampled every 5ms. The phases of the second benchmarks switched faster than the wall measurement sampling rate. RAPL was able to capture these phase switches, while they were not visible in the wall measurements. Khan et al. also investigated the impact of temperature changes. They found a correlation between temperature increase and power consumption. Furthermore, Skylake showed improved performance, reducing the increase in power consumption due to increased temperature. Finally, they looked at the timing of RAPL MSR updates. They found that there is a measurable delay between updates of different registers. They thus indicate that if polling is used to find when registers are updated, the update order should first be investigated. The polling can then be used on the register that is updated last, to ensure all registers have been updated.

Liu et al. [19] produced jRAPL, a library that allows Java developers to access RAPL MSRs that contain energy consumption information within their java code. This can be used to obtain information about arbitrarily chosen code segments, although it remains important to be aware of the update frequency of the MSRs. Pereira et al. [20] proposed an adaptation to Spectrum-based Fault Localization (SFL), which they named Spectrum-based Energy Energy Leak Localization (SPELL). This is a language-independent technique to identify ”hot spots” in code to help developers find where they should focus their optimisation efforts. This technique requires energy consumption information as input. By combining this technique with jRAPL, they performed empirical studies with Java programmes. They found that their technique could reduce the time spent on optimising software for energy consumption and performance by 50%, while attaining energy consumption reductions

of 18% on average.

Beyer et al. [21] developed the “CPU energy meter” tool which uses RAPL to obtain energy consumption information. It makes it easier for users to obtain this information by handling the interaction with the registers. A user can start and stop the tool, obtaining information about the energy consumption during the run, or the tool can be given a program as an argument and it will measure energy consumption while the program is executing. They also integrated their tool into “BencExec”, a benchmarking tool used by researchers and for competitions in the formal methods domain.

During our research, Ournani et al. [22] published a study where they looked at the variability of energy consumption measurements and the influence of multiple hardware settings on this variability. They looked at the influence of the experiment protocol, different CPU settings, the hardware generations and the operating systems. They used RAPL and PowerAPI, which is a tool that also uses RAPL data, to monitor energy consumption. One of the things they found was that disabling C-states, which handles switching CPU frequency based on workload, can significantly reduce variability with low workloads, but has almost no impact at high loads. At high loads, all CPU cores are used which means that no cores are scaled back. Although disabling C-states can reduce variability, it significantly increases energy consumption, since all cores run at the highest setting and are not scaled down if they are idle. They also discovered that pinning processes to cores can influence variability. They found that the best strategy was to pin processes to a single socket, with disabling hyper-threading while pinning to multiple sockets shows a higher variability but slightly lower total energy consumption. Using hyper-threading to pin multiple processes to cores while also utilising multiple sockets showed the worst variability and energy consumption.

2.2 Software energy consumption

Couto et al. [11] developed an approach to rank the efficiency of programming languages. They used solutions for the same problems in different programming languages. As developing such solutions is complex and time-consuming, they used the ”Computer Language Benchmarks Game” project’s repository to obtain implementations. These benchmarks have been used in several research projects. Couto et al. selected 10 programming languages to compare. They decided to use RAPL to perform energy consumption measurements. As RAPL was only usable from C and Java directly, they wrote a small C program that handles the RAPL interaction and starts the implementation to be tested. They verified that this program introduced a small overhead, but that this was insignificant, consistent and negligible. A

later study [12] extended this research to include 27 programming languages. Furthermore, the measured data was extended with data about peak memory usage.

Beyer et al. [21] used the “CPU energy meter” they developed to measure the energy consumption of different software verification implementations at a yearly international competition. An additional green ranking was created that used the information on energy consumption gathered by the tool. They found that the ranking differed considerably from the main score-based ranking, with no overlap in the top 3. Furthermore, they discovered that the winner of the ”green” ranking had an energy consumption two orders of magnitude lower than the worst scoring implementation. This could indicate that there is a lot of potential to reduce the energy consumption of software if developers have access to energy consumption information. By studying two tools in detail, they found that verification tasks with similar execution times showed significantly different energy consumption. This indicates that execution time is not necessarily linked to energy consumption and information on execution time is not enough to draw conclusions for energy consumption.

Pereira et al. [13] used jRAPL to investigate the energy consumption of different Java Collection Framework implementations. They analysed different Set, List and Map implementations by measuring the energy consumption of the available methods with different collection sizes. They found that there was no one best implementation, but that it was possible to make a choice that reduced energy consumption based on the methods that are used in the software. This does mean that if the software is changed, a different collection implementation might use less energy However, it is not always trivial to change the collection that is used, as they are not all equivalent. They also developed a tool that uses static analysis to find the use of collection classes in a Java project, jStanley [23]. This tool then uses information about energy consumption from their previous work to suggest a more energy-efficient alternative. Hasan et al. [24] also investigated the energy consumption of Java classes. Their tests were performed on a Raspberry Pi with an Arduino board collecting the energy consumption measurements. They used smaller collection sizes and a different set of collections, with some overlap, compared to the work by Pereira et al. They found that collections with more elements showed larger differences in energy consumption. They also found that the type of element used in a collection can impact energy consumption. Primitive types increased energy consumption, most likely because extra operations are required to box primitive types before they can be used with collections. They also found that when execution time and energy consumption increased, power use showed no change. This indicates that the extra energy consumption is caused by the increased time spent. Pinto et al. [25] also looked at the energy consumption of Java Collection Framework implementations, focusing on thread-safe implementations. They investigated the energy

consumption of different methods and the impact of the number of threads on energy consumption. During their experiment, they found that calculating upper bound limits for loops in each iteration consumed twice as much energy compared to calculating the limit once and storing it in a variable for a specific collection. They do warn that operations that change the length of the collection require calculating the length in every iteration for correct performance. By using the information obtained from their experiments they managed to half the energy consumption of their micro-benchmarks while applying the changes to real-world benchmarks improved energy consumption by 10%. The impact of for-loop syntax was also studied by Tonini et al. [26] as one of the practices suggested by Google to improve performance on android. They focused on iterating arrays, also finding that calculating length in every iteration increases energy consumption. They also investigated the for-each syntax, finding that it can increase energy consumption even more.

Gabriel Lima et al. [27] [28] investigated the impact of data structures in Haskell on energy consumption. For sequential programs, they found that execution time and energy consumption were strongly correlated. Faster execution times also lead to a reduction in energy consumption. They also looked at concurrent programming constructs. While they used micro-benchmarks to analyse the energy consumption of sequential data structures, they used benchmarks from Computer Language Benchmark Game (CLBG) and Rosetta Code as well as some self-developed benchmarks to analyse the energy consumption of concurrent programming structures. They found that it is possible to obtain significant energy reductions with small changes to the code, such as changing the data type of a variable or using a different fork method. Furthermore, they found that execution time and energy consumption are not correlated in concurrent programs. Several programming changes reduced the execution time while increasing energy consumption. Finally, they found that using more capabilities, virtual processors in the Haskell run-time system, than the number of cores available on the CPU can drastically increase execution time and energy consumption. They found that most benchmarks also showed a decreased performance when setting the number of capabilities equal to the number of virtual cores provided by Intel's hyperthreading. Only one benchmark showed improved performance.

Melfe et al. [14] elaborated on the study by Gabriel Lima et al., focused on studying the energy consumed by DRAM by different data structures in sequential programs. They found that DRAM energy consumption was also correlated with the execution time. They also found that DRAM was responsible for approximately 15% to 30% of the total energy consumption. Melfe et al. also looked at the impact of Glassgow Haskell Compiler (GHC) optimisation options on execution time and energy consumption. They found that optimisations reducing the execution time also

decreased energy consumption. However, they encountered some cases in which the optimisations increased execution time and energy consumption. So the relation between execution time and energy consumption is maintained but the optimisation options do not guarantee an improved performance based on execution time and energy consumption.

A different study by Melfe et al. [29] compared the impact on energy consumption of lazy evaluation of Haskell data structures to strict evaluation. They use micro-benchmarks to analyse three map implementations, using both lazy and strict evaluation. They once more found that execution time and energy consumption are related. In most cases, strict evaluation showed a reduced execution time and energy consumption while lazy showed a better performance in specific cases. Their analysis is limited to micro-benchmarks. They had plans to analyse more complex programs to find out if these findings could be generalised.

Chantarasathaporn. [30] analysed programming strategies in C#. They looked at the execution time of different strategies as a proxy for energy consumption. They compared choices that can be functionally equivalent such as using a struct or a data-member-only class, static or dynamic attributes and methods and method and variable accessibility. Some of the choices showed significant differences in execution time while others showed no significant difference. For example, they found a protected variable was slower than a private or public variable by 40% while the accessibility of methods showed no difference. As they created small pieces of code to specifically test their alternatives, it is unclear how this translates to actual software. As they only measured execution time, it is also not certain that the differences in execution time also indicate differences in energy consumption. However, the fact that differences were observed indicates that there is a potential for different energy consumption behaviour if different programming choices are made.

Litke et al. [31] applied five different design patterns to embedded C++ code. They measured the energy consumption of the code with the design pattern and compared it to the energy consumption of the code without the design pattern. For one of the design patterns (Observer) they found a significant increase in energy consumption compared to the code without the pattern. The other patterns showed no difference in energy consumption. It appears they made use of small code examples to perform their tests, so it is unclear if their findings can be generalised. Bunse et al. [15] also investigated the impact of design patterns, focusing on mobile Java apps. They selected six design patterns and created applications with and without the design patterns. They found that three of the patterns showed no difference in energy consumption (including Observer) while two showed a small increase in energy consumption. For one pattern (Decorator), the energy consumption more than doubled. As the applications were specifically developed to test the impact of the

patterns, it is unclear how a design pattern might impact the energy consumption of more complex software. Sahin et al. [32] used an FPGA to investigate the energy consumption of design patterns. They obtained sample code for 15 design patterns. For some design patterns, the code with the pattern applied reduced the energy consumption, for some the energy consumption was not significantly different while for others it was increased. Interestingly, they found that applying the observer pattern increased the energy consumption by 60% while the decorator pattern increased the energy consumption by 700%. Feitosa et al. [33] investigated the impact of Design patterns in two non-trivial Java software systems. They detected the uses of design patterns and manually created a second version of the software, replacing applications of design patterns with alternative solutions. Measuring the energy consumption of both versions, they found that the alternative versions showed reduced energy consumption. By further analysing the instances where the patterns were replaced, they found that the reduction in energy consumption was smaller for more complex code. This could have implications for the generalisability of results from research with sample code. The differences in energy consumption observed by such research could be reduced when the patterns are applied to non-trivial programs. However, the implementation of design patterns in non-trivial programs is likely to differ, as developers make different choices. This makes it difficult to repeat the research using such systems.

Noureddine et al. [34] investigated if and how the energy consumption impact of design patterns could be reduced. They focused on the Observer and Decorator pattern, as research has shown that they significantly increase energy consumption. They created small transformation rules aimed at reducing the energy consumption of these patterns, with the eventual goal to apply them automatically. By applying these transformations to existing software, they found that the overall energy consumption was decreased by 4% to 25%. More savings were attained in software that made more extensive use of the design patterns. Such transformation rules can help developers retain the advantages of design patterns while reducing their impact on energy consumption. Furthermore, it shows that programming choices can have an impact on overall energy consumption.

Agosta et al. [35] investigated Java programs for the financial sector and if memoization could have an impact on the energy consumption of such programs. Memoization is a technique were calculated results are stored in memory along with the input and function that created the results. If a calculation is encountered at a later stage, the results can be retrieved from memory instead of repeating the calculation. As storing results in memory incurs new energy costs, it needed to be investigated if the process could attain overall energy savings. They used bytecode analysis to identify pure functions, functions that do not have side effects and are determin-

istic. Although functions that create objects are sometimes seen as pure in Java, these are also excluded, as the created object has to be unique for every invocation, preventing the use of memoization. Based on a set of empirically tuned criteria, candidates for memoization are selected. These functions are wrapped to perform memoization. If a new calculation is performed, a trade-off function decides if the results should be stored, for example, based on the available memory. Agosta et al. created a performance model to calculate the effectiveness of using memoization. This is based on the difference between performing a calculation and reading a value from memory and the hit rate of the stored values. The hit rate is in turn based on the variance of the parameters and the available memory. They applied this process to several open-source financial functions and a part of a well-known benchmark. Several executions are necessary before the memoization version stabilises, as the lookup table needs to be filled and stabilised. They found that the memoization version reduced execution time and energy consumption in all cases. For two of the four tested functions, the energy consumption was reduced by several orders of magnitude. This shows that it is not only possible to reduce energy consumption by making changes to the syntax, but also by reworking the overall process. Of course, this is a much more complicated task requiring a lot of in-depth knowledge.

Chapter 3

Methodology

This chapter describes the methodology we used in the experiments. We describe the hardware and the benchmarks that were used, the measurement approach, the variables that were measured with a justification for these variables and how the results were analysed.

3.1 Approach

There are still a lot of unknowns in the field of energy consumption of software. Especially for the programming language that we focus on, C#, there is a lack of evidence. Thus, we perform an empirical study to obtain evidence so that we can answer our research questions. We document factors that might have an impact on the observed results in this chapter. Hopefully, this will make it possible for any future research to replicate the results we observed.

All energy measurements are performed with existing code, requiring no new implementation on our part. Some small changes had to be made to remove compiler warnings and errors. To automate the testing process, some shell scripts were created. Furthermore, we created some Visual Basic (VB) macros to assist us in analysing the results of the measurements.

3.2 Hardware

The main platform on which the experiments were performed is a Lenovo P1 gen 2, which contains a 9th generation (Coffee Lake) Intel Core i7-9750H CPU. This CPU has 6 cores and uses a 16GB RAM (DDR4-2666, 2 SoDIMM). It uses a Dual boot, Windows 10 and Ubuntu 18.04.04, with the experiments performed under Ubuntu.

To check if the patterns observed in the experiment results are unique to this hardware system, tests were also performed on a multitude of other systems. Tests

were performed on a Dell Precision M2800, an HP Elitebook and a Dell Precision M2800, the specifications can be found in Table 3.1

Model	CPU architecture	CPU Model	Cores	RAM
Lenovo P1 gen 2	9th generation	i7-9750H	6	16 GB
Dell Latitude E5570	6th generation	i5-6300U	2	8 GB
HP Elitebook 840 G3	6th generation	i5-6300U	2	8 GB
Dell Precision M2800	4th generation	i7-4710MQ	4	8 GB

Table 3.1: Hardware systems

All tests not executed on the Lenovo P1 were executed using a USB with Ubuntu 18.04.04 installed and using a live boot, without installing Ubuntu on the system. These systems were available for a short time and it was not viable to install Ubuntu on the system.

3.3 Benchmark

The experiments that tested the influence of hardware and compiler settings on the energy consumption of software used code from the Computer Language Benchmark Game (CLBG), in particular the C# implementations. The CLBG initiative was created to compare the performance of solutions to problems written in different programming languages. It includes a framework for running, testing and comparing similar implementations. Solutions have been gathered in many different programming languages, but for direct comparisons, the solutions have to follow a given algorithm and specific implementation guidelines. Although it was created to compare performance, it has recently also been used to evaluate energy consumption [11] [12] [27]. The code used in our experiments was retrieved from the repository published by Pereira et al. [12] on the accompanying website[1]. Some changes had to be made to be able to execute the code and perform the measurements. These changes concerned syntax changes caused by updates to .NET Core and Python. The code was compiled with .NET SDK 3.1. In Table 3.2, an overview of the benchmarks that are used in this research is given, with a short description and an indication if they make use of parallel programming.

[1] https://sites.google.com/view/energy-efficiency-languages

Name	Description	Parallel
Binary-trees	Allocate and deallocate many binary trees	✓
Fannkuch-redux	Indexed-access to tiny integer-sequence	✓
Fasta	Generate and write random DNA sequences	✓
K-nucleotide	Hashtable update and k-nucleotide strings	x
Mandelbrot	Generate Mandelbrot set portable bitmap file	✓
N-body	Double-precision N-body simulation	x
Pidigits	Streaming arbitrary-precision arithmetic	x
Regex-redux	Match DNA 8-mers and substitute magic patters	✓
Reverse-complement	Read DNA sequences - write their reverse-complement	✓
Spectral-norm	Eigenvalue using the power method	✓

Table 3.2: Benchmarks description

3.4 Tested variables

The experiments around the influence of hardware settings on energy consumption focused on two CPU hardware settings. These settings are Hyper-threading, a setting on Intel CPUs that allows the operating system to address one physical core as two virtual cores, and the CPU scaling governor. Hyper-threading is a setting that can be enabled/disabled in the BIOS, while the governor can be set to power-save or performance from the command line. The default setting is Hyper-threading enabled and the governor set to power-save. These settings were chosen as this research is focused on the availability of energy consumption information for general developers. Changing the governor can be easily automated while changing a BIOS setting might be more complicated, as BIOS settings might be locked on company laptops. More complicated settings were not tested, as this requires more specific knowledge on the part of the developer to understand what they are doing. If such knowledge is not available, changes might damage a system.

The experiments around compiler settings tested different combinations of com-

piler settings offered by .NET Core[2]. C# uses a just-in-time (JIT) compiler and .NET Core offers options to adapt the compilation behaviour. These options produce sub-optimal code in a shorter amount of time. If a method is used often, the code can be replaced by an optimised version to improve the execution behaviour. The settings are Quick JIT, Quick JIT for loops, ReadyToRun and Tiered compilation. Quick JIT compiles methods without loops more quickly but without optimisations. This can reduce the startup time of a program but can reduce overall performance. This setting is enabled by default since .NET Core 3.0. Quick JIT for loops applies Quick JIT to methods that contain loops. This may improve startup time but can cause long-running loops to get stuck in less-optimised code. This setting is disabled by default. ReadyToRun is a form of ahead-of-time compilation. It improves startup time by reducing the amount of work that the JIT compiler has to perform. The created binaries when using ReadyToRun are larger, as they contain both the intermediate language code and the native code. Furthermore, it has to be compiled for a specific runtime environment. Tiered compilation starts with first-tier code from Quick JIT or ReadyToRun and will then work on optimising this code in the background. Tiered compilation is enabled by default since .NET 3.0. Throughout this paper, these settings will be abbreviated as follows:

Q Quick JIT

L Quick JIT for loops

T Tiered compilation

R ReadyToRun

The experiments tested Q, QL, QT, QLT, T, R, RT, QLTR. According to the documentation, enabling Quick JIT for loops while Quick JIT is disabled has no effect, thus this combination was omitted. Furthermore Tiered compilation without Quick JIT or ReadyToRun should behave the same as disabling Tiered compilation. The documentation is also unclear on what happens when combining Quick JIT and ReadyToRun with Tiered compilation.

3.5 Measurement Tools & Methodology

Energy consumption information was obtained using the Intel Running Average Power Limit interface. This is an easy to use method available on Linux systems with Intel CPUs with a Sandy Bridge architecture or newer. The energy consumption measured by RAPL has been proven to be accurate. It should be noted that it is

[2] https://docs.microsoft.com/en-us/dotnet/core/run-time-config/compilation

limited to energy consumption of the CPU, including the cores, on-chip GPU, DRAM and on Skylake, more on-chip systems, as can be seen in Figure 2.1.

For the experiments run on the Lenovo P1, the Psys domain is used to compare the results of different settings. This domain tracks the entire CPU and should thus provide the most complete view of the energy consumption. For the HP and Dell Precision, Psys results are not available and as such the package domain is used instead.

The Lenovo P1 contains 2 GPUs, an Intel GPU on the CPU and an NVIDIA GPU separated from the CPU. On Ubuntu, it is possible to indicate which GPU should be used. The results of running the test using either GPU were compared to verify how the GPU impacted the energy consumption results. It was found that while measuring 10 executions of a benchmark, 2 or 3 executions showed GPU energy consumption. For some benchmarks, every execution showed GPU energy consumption, however, this was a negligible amount. As such, we decided to execute the tests while using the NVIDIA GPU to remove the possibility of different processes using the GPU impacting the energy consumption results.

During the benchmark tests, a small C script, created by Pereira et al. [12], handles the interaction with the RAPL registers and starting an individual test. While testing energy consumption while running idle, it was discovered that the C method, `system(command)`, used to call the code under test introduces an overhead of 2 to 5 milliseconds. This was measured by comparing the results of the original C script calling code that sleeps for 10 milliseconds to an edited C script that directly sleeps 10 milliseconds. The execution time of individual benchmarks ranges from .3 to 20 seconds, thus the overhead of 2 to 5 milliseconds was deemed acceptable.

A separate Python script is responsible for calling the C script for every benchmark test that should be run. Every benchmark that is used is measured 10 times. The Python script then waits 5 seconds before measuring the next benchmark. For the tests on live Ubuntu, this Python script was replaced by a shell script. This change was made because the live Ubuntu session ran out of memory during tests, making it impossible to perform the tests. We verified that this change did not change the energy measurement results, and found no significant difference.

When executing a benchmark, the initial runs show increased variability. Thus a benchmark is executed 10 times to reduce the impact of this initial variability. The results of the benchmarks are then stored in a CSV file. The set of benchmarks is then executed a total of five times. This is done to check if benchmarks show similar energy consumption in different runs. One execution of the benchmarks takes about 15 minutes, meaning that the tests for one variable take approximately 1 hour and 15 minutes to 1 hour and 30 minutes. The initial set of tests was executed 10 times. As the results of these additional tests did not significantly impact the results, 5 sets

of tests were chosen to keep the execution time manageable.

To automate the process of running tests, a shell script has been created that compiles the code, starts a set of tests, stores the result of the tests and then starts another set of tests. It will execute the set of tests a total of 5 times. A further automation step was made by creating a repository with different compilation settings on different branches. Another shell script is used to switch to a repository branch, execute the tests and continue to the next branch.

Since the RAPL MSRs store the energy consumption in 32 bits and are continuous, they will sometimes overflow. The rate at which they overflow depends on how much energy is consumed. Khan et al. [9] calculated that a Haswell machine using 84W would cause an overflow every 52 minutes and argue that sampling the registers every 5 minutes should be sufficient to detect overflows. During this research, the registers are checked with a much higher frequency. All overflows occur because a register is close to the overflow value at the start of the test and will result in a negative value in the resulting CSV. Since any negative value indicates an overflow, these can easily be (manually) detected and excluded from the data used to compare results.

Chapter 4

Data Analysis

In this chapter, we discuss the different methods we used to analyse the results of the experiments. We used multiple methods to analyse and visualise the data before we settled on the approach we use to visualise them in this thesis. The analysis and visualisation are discussed in Section 4.1. In Section 4.2, we discuss how we performed the analysis. The statistical analysis we used to check if observed differences are significant is discussed in Section 4.3

4.1 Analysis & Visualisation

Every run of a set of tests produces a CSV file. This CSV file contains the results for 10 run sequential runs of each benchmark. It contains the values for each RAPL domain, as well as the CPU temperature at the start and the end of a benchmark execution and the total execution time. An example of such a CSV file is visualised in Appendix A. When automatic tests are executed, a name for the CSV files is required as a parameter. The CSV files for the five executions of the benchmark set will then automatically get this name along with a number from 1 to 5. We created a VB macro that takes the location and name of the CSV files without the number. It will then import all 5 CSV files into individual Excel worksheets.

In our first approach to analysing the results, we used Excel functions to calculate the average value for each benchmark, while keeping each set of benchmark executions separate. Standard deviation, the difference between the maximum result and the average value and the difference between the minimum result and the average value were calculated in a similar fashion. We used a visual comparison to see if the results showed an acceptable variation and if we could already observe any trends. We then elaborated on this approach by calculating the average and standard deviation for each benchmark but combining multiple executions of the set of benchmarks. This allowed us to test if multiple executions of the set of benchmarks

showed an increased variation or if the variation was stable. We also calculated a second average, using the last four results of the ten results for each benchmark. This was calculated to see if the initial startup or caching could have an impact on energy consumption.

We then took a slightly different approach in order to be able to make proper comparisons. We calculated a combined average over the five executions for each benchmark, focused on the PKG and Psys domain. We then calculated the standard deviation and the standard error so we could calculate a confidence interval. This was then used to create graphs to visualise the data. Bar graphs were used to indicate the average energy consumption with error bars used to visualise the confidence interval. An example of such a graph can be found in Figure B.1. Comparing these graphs allowed us to observe trends in energy consumption for different compiler options.

Since there were differences in execution time and energy consumption, we wanted to know if we could learn something from calculating the energy consumed per time unit. For each execution of a benchmark, we calculated a new value by dividing the energy consumption reported according to the Psys domain by the execution time. This gave us the Joules consumed per millisecond. For this, we also calculated average values, standard deviations and standard errors and created charts for each benchmark. An example of such a chart can be found in Figure B.2.

However, as these methods require an individual chart for every benchmark, it is difficult to easily compare results for multiple variables. To make such comparisons easier, we combined the results of the different benchmarks. This was done by summing the averages for each benchmark to obtain an overall average. The overall standard deviation was calculated according to Equation (4.1)

$$\sigma_{total} = \sqrt{\sigma_a^2 + \sigma_b^2 + \sigma_c^2 + ...} \tag{4.1}$$

The overall standard error was calculated according to Equation (4.2)

$$SE = \sqrt{\frac{\sigma_a^2}{N_a} + \frac{\sigma_b^2}{N_b} + \frac{\sigma_c^2}{N_c} + ...} \tag{4.2}$$

By calculating these values for the Psys domain and the execution time, we created a two dimensional graph to compare different settings. An example of such a graph can be found in Figure B.3.

While looking into a newly published article [22], we encountered the use of box-and-whisker plots to visualise and compare energy consumption measurement results with different variables. A box-and-whisker plot visualises groups of data through their quartiles. The line in the box represents the median, the top of the box represents the median of the upper half of the dataset and the bottom of the

box represents the median of the lower half of the data. The lines, or whiskers, represent the minimum and maximum value, while any dots above or below these whiskers are values classified as outliers. An example of a box-and-whisker plot can be found in Figure 5.1. We found that such charts better represent the total range of the measurement results, allowing for a better comparison of results. As these charts only have one axis, we represent the data with two box-and-whisker plots, one for the energy consumption, either PKG or Psys, and one for the execution time. Excel can automatically create these plots from a dataset.

4.2 Process

To reduce the effort required to perform the analysis we are using, we created a VB macro. We created a template worksheet in an Excel workbook. This template contained all the formulas and charts that we wanted to obtain based on the data from five sets of benchmark executions. After the CSVs have been imported into individual Excel worksheets, in a separate workbook, the macro can be used to retrieve the data from these worksheets and correctly place it in a copy of the template worksheet. The formulas and charts will then update based on the new data. Charts that combine results for multiple variables are still created manually, based on which worksheets should be included.

Initially, we encountered an issue when copying charts from one worksheet to another. The chart will still use the data from the original worksheet. We created a macro to automatically replace the references to the worksheet based on an input parameter. This turned out only partially successful for the charts with error bars, VB can be used to access the data range for the bars themselves. However, the data range for the error bars is not exposed to VB and it is thus impossible to change the worksheet reference with a macro. This problem is avoided by using a template sheet. When copying an entire worksheet, the new charts will automatically use the identical data range on the new worksheet.

4.3 Statistical Analysis

To ensure that the differences and trends we observed are significant, we performed a statistical analysis. Firstly, we investigated how the data is distributed. We used the Shapiro-Wilk test to check if the data is normally distributed. The Shapiro-Wilk test tests the null hypothesis that a given sample $(x_0, x_1, ..x_n)$ came from a normal distribution. We performed this test, taking the results for one benchmark as the samples. For some benchmarks, the results indicated that the null hypothesis can

be rejected, meaning that the samples are not from a normal distribution. For other benchmarks, the results did not support rejecting the null hypothesis. Furthermore, after we repeated this test for the results from runs with different compiler settings, the results of the Shapiro-Wilk test were not consistent. While the null hypothesis could be rejected for a benchmark for one compiler setting, this was not true for all compiler benchmarks. For all benchmarks, there was at least one setting where the null hypothesis could be rejected and at least one where it could not be rejected.

Based on the Shapiro-Wilk test results, we decided to use the Wilcoxon rank-sum test. This is a nonparametric test that can be used to test if two independent samples were selected from populations with the same distribution. We selected this test as we cannot treat our results as being from a binary distribution. We performed this test for individual benchmarks. Similar to what we did for the analysis described in Section 4.1, we created a VB macro that retrieves the results for each benchmark from the worksheet they have been imported into and place it in the correct location on a copy of a template worksheet. The test requires the results for two variables, grouped per benchmark. Analysing the results of the Wilcoxon rank-sum test, we found that for some compiler settings, the energy consumption of some benchmarks was significantly lower, while for others it was significantly higher. Therefore we decided to report the results of the Wilcoxon rank-sum test along with the average energy consumption. This gives a better indication of the actual impact, as some benchmarks have an energy consumption that is magnitudes of order smaller than other benchmarks.

Chapter 5

Measurement

In this chapter, we discuss how developers can perform energy consumption measurements. We discuss how developers can use RAPL for energy measurements and how they can reduce the variability of such measurements in Section 5.1. We also analyse the idle consumption of the system we used and how this compares to the energy consumption under load in Section 5.2. In Section 5.4, we give an answer to RQ1.

5.1 Energy Measurement

It should be noted that to be able to use RAPL to measure energy consumption, a Linux distribution is required. Accessing registers requires special access (Ring-0). To achieve this in a Windows distribution, a special driver needs to be used and no such driver is publicly available for Windows at this time.

On Linux, there are multiple methods that use RAPL to obtain energy consumption information. The first method is to manually read the machine-specific registers. This requires root access and knowledge about the location of these registers. The registers should be read once at the start of the measurement and once at the end of the measurement. The difference between these values can be used to calculate the energy consumption. The value read from the register is stored in a unit of measurement defined by Intel. It needs to be multiplied by a factor stored in a different register to obtain the energy consumption in Joules. This is done to increase the capacity of the register, as Intel uses 32 bits of a 64 register which would overflow more frequently without this conversion. As RAPL starts tracking energy consumption once a computer boots, no action is required before the registers can be accessed. A second method is to use a tool that handles the interaction with the registers and provides the results to the user. An example of such a tool is the CPU energy meter developed by Beyer et al. [21]. Perf is another tool that can be used.

It is provided on Linux by default and can be used to track a multitude of system information, including RAPL on Intel systems. Performance API is a similar tool that can provide information on a range of system information and has been extended to include RAPL information [36].

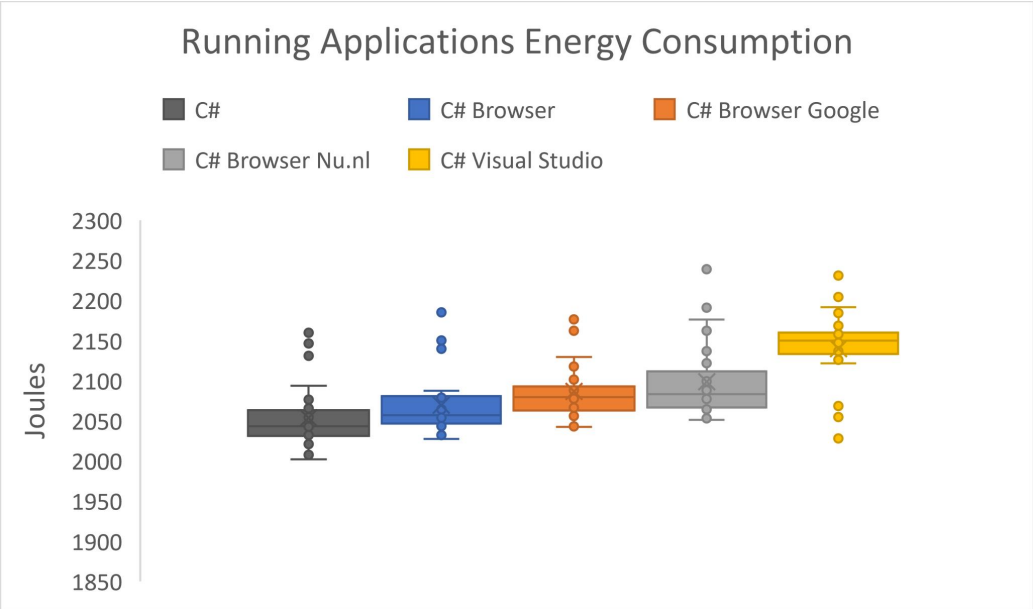

Figure 5.1: Energy consumption with different applications running in the background

RAPL tracks the energy consumption of the different domains without tracking which process is actually using those domains. As such it is important to perform measurements while minimising the number of other processes that are running. Tests were performed while running other applications. In Figure 5.1, the results of a measurement without other applications are compared to the results of running a Firefox browser without a web page, with `google.com`, with `nu.nl` and with Visual Studio minimised in the background. The limited activity already showed increased usage and having web pages open further increased consumption and variability.

Initial measurements showed stable results with limited variability. However, over the course of the research project, some tests were performed again, showing different results. The variability still overlapped but combining the results would show an increased total variability. As tests were performed over a longer period of time, on different locations and with different external temperatures, these factors could be responsible for some of this increased variability.

We also found that it should be ensured that a system does not fall asleep while executing the tests. If this occurs, the results will be unreliable, as it is unclear what happens, which processes are stopped and which continue. Furthermore, waking up introduces extra costs. This can be avoided by changing the display settings to

always stay on. When performing tests on a laptop, tests should be executed while connected to an external power source. If tests are performed while relying on the battery, the results will be impacted by the remaining charge in the battery, as the operating system will take additional power-saving measures when the remaining power passes certain thresholds.

5.2 Idle energy consumption

To get a baseline of the energy consumption, we measured energy consumption while running idle. We performed these measurements with different lengths, namely 10 milliseconds, 100 milliseconds, 1 second (1000 milliseconds) and 10 seconds. In this experiment, we measured the energy consumption 500 times. These 500 measurements were repeated 5 times, except for the 10 second measurements, for which the set of 500 measurements was executed just once.

During these measurements, we discovered the overhead cost caused by the C function `system(command)`, which was also mentioned in Section 3.5. The results for measuring 10 milliseconds idle showed an execution time of 11 to 15 milliseconds, while the results for 100 milliseconds showed an execution time of 101 to 105 milliseconds and the results for 1000 milliseconds showed an execution time of 1001 to 1005 milliseconds. We noticed this constant offset and some changes to the code running the test allowed us to pinpoint the cause. This overhead has a significant impact on the shorter measurements. Repeating the measurements for 10 milliseconds while excluding this function gave results with an average execution time of 10.5 milliseconds.

Table 5.1: Average Energy consumption while running idle. The last column shows the results of removing a function introducing overhead costs.

Duration	10ms	100ms	1000ms	10s	10ms*
Average Energy Consumption	0,142J	0,949J	8,582J	85,69J	0,0927J
J/ms	0,0142	0,0095	0,0086	0,0086	0,0093

The results for the energy consumption while running idle are very stable with limited variability. The average energy consumption can be found in Table 5.1. The last column provides the average consumption for the experiment where the `system(command)` function was excluded. This shows that this call has a serious impact on short measurements while running idle. From these results, it is also possible to see a linear trend between execution time and energy consumption. The second row in the table shows the energy consumed per millisecond. This value is

pretty similar, with a slightly lower value for the longer execution times. This can be explained by the fact that energy is measured once per execution, thus longer idle times will reduce the impact of any overhead costs incurred.

We compared these values to the energy consumption while performing the benchmark. The energy consumption per millisecond while executing the benchmarks ranges from 0,03 J/ms to 0,07 J/ms, depending on the benchmark. From these values, we can conclude that the baseline costs are at most 30% of the energy consumption while running a benchmark test. Most of the energy consumption is caused by the code being executed.

5.3 Validity

There are several factors that can influence the validity of the results presented in this chapter. First of all, the applications that were left running while performing the benchmark measurements were left idle. If a user were to continue to use a system while a measurement is being performed, it is likely that the energy consumption will show more variability. While the applications are idle, the scheduler should give priority to the active benchmark. If the system is being used, it is likely that the benchmark will get less time from the scheduler. As it is not trivial to perform multiple automatic measurements with comparable user input, it was decided to continue with an idle system without any other idle or active applications.

Secondly, it is not yet clear how idle energy consumption relates to baseline consumption or consumption under load. [22] et al. found that systems showed an increased variability while idle compared to running a task. This indicates that there could be a difference in energy consumption behaviour of a system under load compared to while running idle. So, while we can conclude that the energy consumption while executing a task is higher than while running idle, it is not possible to make further conclusions.

5.4 Conclusion

Based on this information, we can give an answer to RQ1: How can the energy consumption of software systems be measured?

Although we saw a limited impact from leaving other applications idle, it is best to reduce the number of applications that are open/active. Furthermore, it should be ensured that a system executing tests is connected to an external power source and that it does not change the operating state while executing the tests, such as shutting off the display, going to a lock screen, or falling asleep.

There are several tools that can help a developer obtain energy consumption measurements using RAPL. If the software being tested has a short execution time, such tools could have a significant impact on the measured results. It is preferable to execute tests with a longer execution time, to reduce the impact of any overhead. Furthermore, it is advisable to perform multiple measurements, as there is always some variability in the results. If you want to compare results from two or more tests, it would be best to (automatically) run all tests in one session. Multiple sessions might introduce additional variance.

Chapter 6

Hardware Settings

In this chapter, we discuss the impact of different hardware settings, both on the energy consumption of an idle system in Section 6.1 and on the energy consumption of C# software in Section 6.2. As mentioned in Chapter 3, this concerns the Hyper-threading setting and the CPU governor. In Section 6.5, we give an answer to RQ2.

6.1 Idle consumption

The energy consumption measurements of an idle system discussed in Section 5.2 were also performed with the different hardware settings. The measurements for 100 milliseconds of idle time can be found in Figure 6.1 and the measurements for 1000 milliseconds of idle time can be found in Figure 6.2. It can be seen that the energy consumption in performance mode is slightly higher than the energy consumption in power-save mode. This is to be expected, as the CPU is running at a higher frequency in performance mode. However, the difference is very limited. Producers of hardware have put a lot of effort into reducing energy consumption while running idle.

There are a number of outliers visible in both charts, but the outliers in Figure 6.2 show a large variation. It should be kept in mind that these charts use 2500 data points and that there are 5 to 10 outliers. The outliers for the power-save mode with Hyper-threading disabled came from a cluster of sequential data points. The two most extreme outliers for the performance mode with Hyper-threading disabled are also two sequential data points. It seems likely that the system was interrupted by a different process, leading to increased energy consumption. Finally, the first results of a set of 500 measurements are also slightly higher than the majority of the results. This is likely due to the system settling into a reduced operating level.

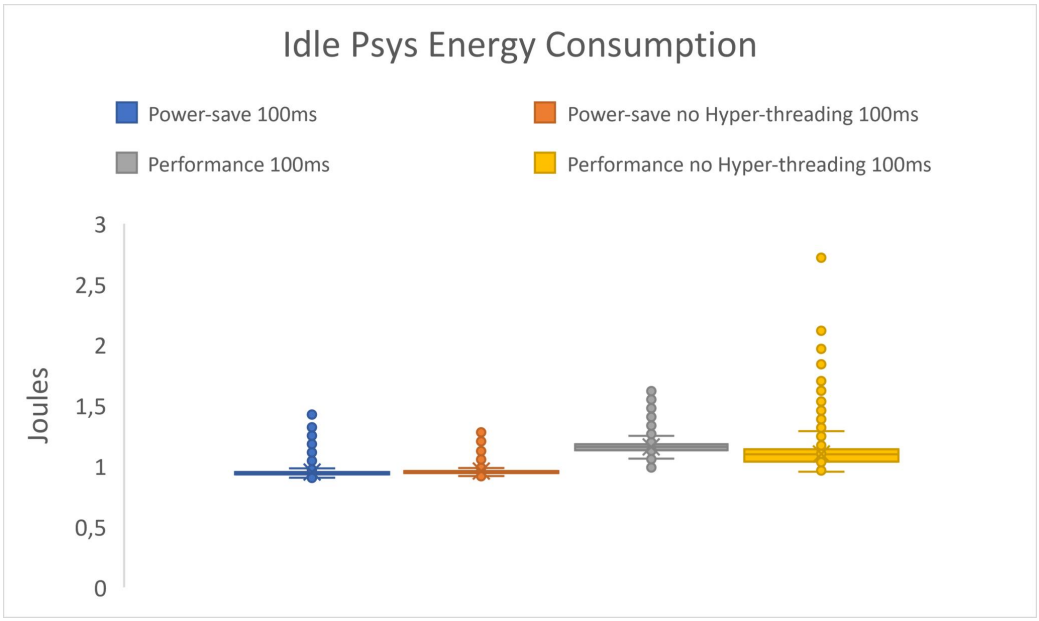

Figure 6.1: 100ms Idle Energy consumption

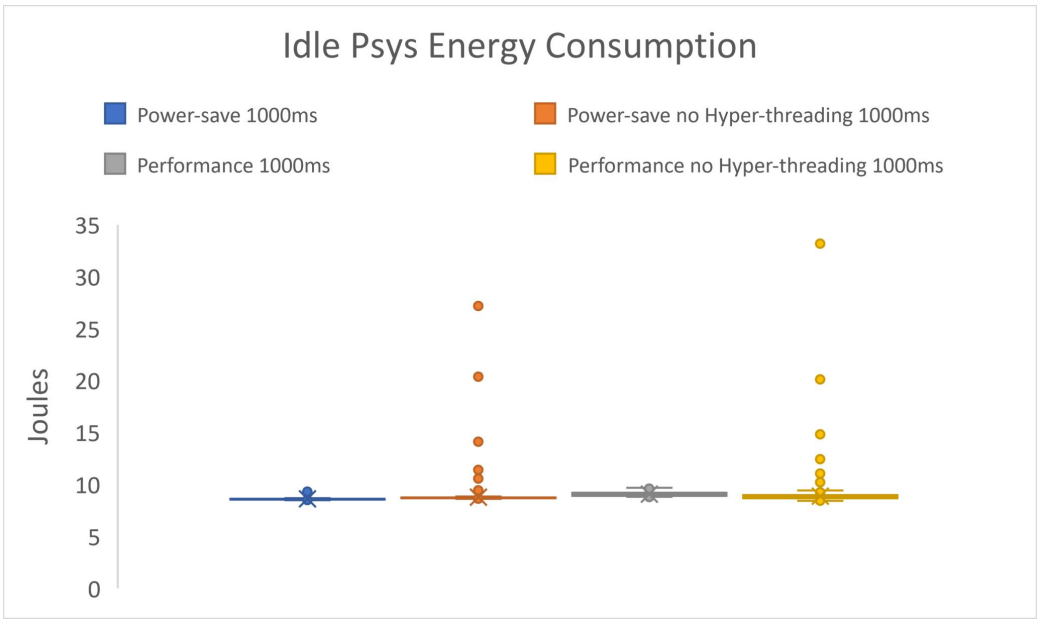

Figure 6.2: 1000ms Idle Energy consumption

6.2 C# Benchmarks

The results of the initial measurements of the benchmarks with varying hardware settings can be found in Figure 6.3 and Figure 6.4. It shows that setting the governor to performance mode can slightly increase the energy consumption while the execution time decreases. Disabling Hyper-threading shows a stronger increase in energy consumption combined with shorter execution time. Disabling Hyper-threading and

setting the governor to performance shows the strongest increase in energy consumption and a decrease in execution time. Furthermore, disabling Hyper-threading reduces the overall variability of the results.

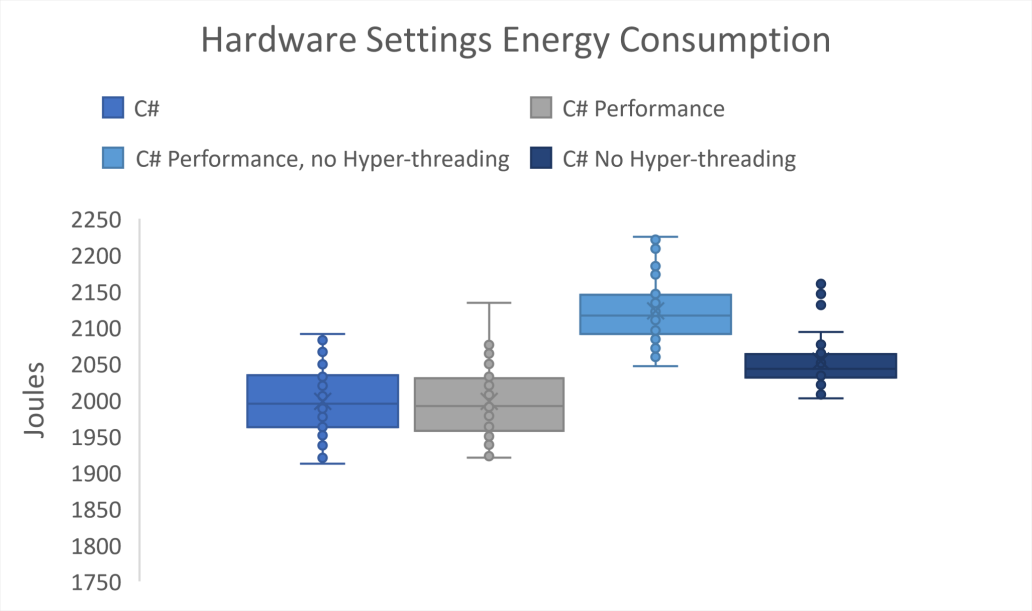

Figure 6.3: Energy consumption

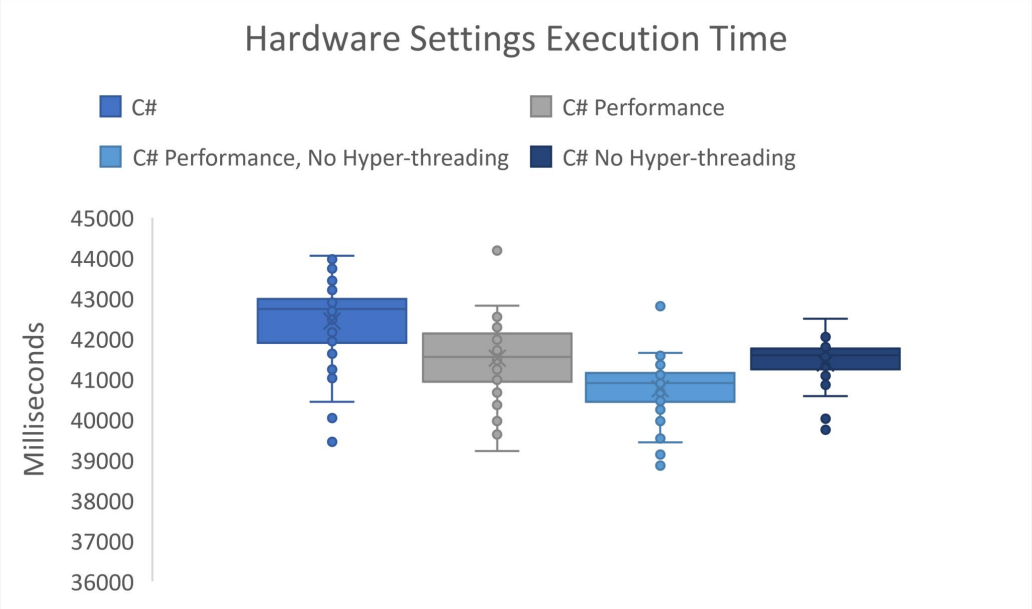

Figure 6.4: Execution time

However, when measurements were repeated at a later time, disabling Hyper-threading showed different results. The results can be seen in Figure 6.5 and Figure 6.6. It still reduces the variability of the results but the energy consumption was

reduced instead of increased. The difference between these sets of measurement results is the fact that in the initial experiment, the compilation files were not changed after applying the hardware setting change. Furthermore, all tests showed an overall decreased execution time while no such general change is present in energy consumption. For the second measurement results, the compilations are discarded at the start of the test and the code is compiled anew. This seems to indicate that the C# compiler adapts to the hardware settings at the time of initial compilation. Furthermore, it does not automatically detect that these settings have changed. This could mean that when hardware settings on a system are changed in order to improve energy consumption if the software is not correctly recompiled, the desired results are not obtained.

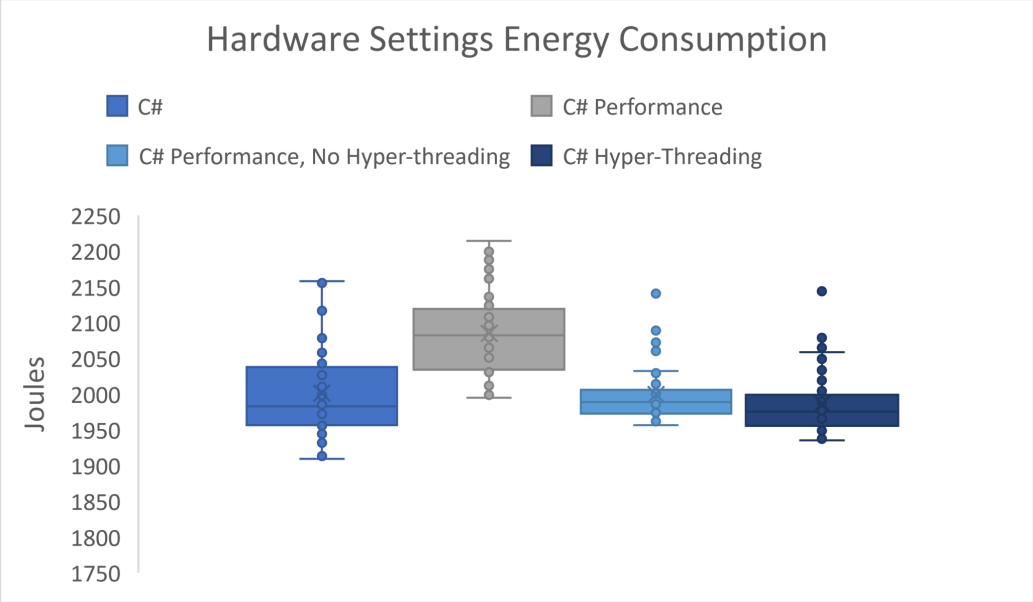

Figure 6.5: Energy consumption

6.3 Comparison to related work

The work published by Ournani et al. [22] similarly looks at the impact of hardware settings on energy consumption. They also measured the energy consumption using RAPL in combination with PowerAPI, which builds a model on top of information from system counters to provide process specific energy consumption information. They performed their research on server nodes, which provide two core sockets, while this research uses laptops which provide a single core socket. Ournani also made use of a different set of benchmarks to perform their tests.

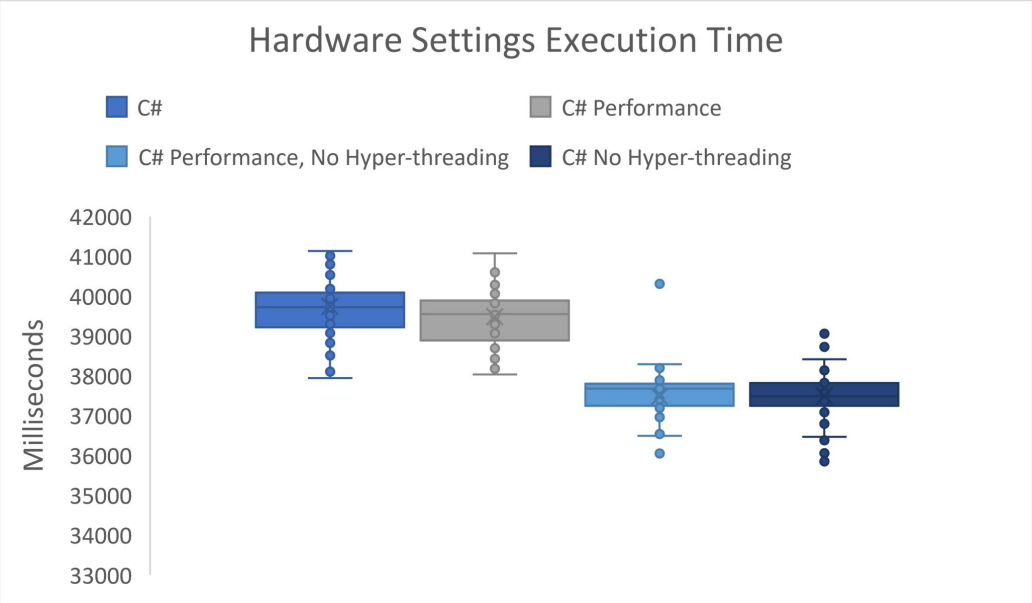

Figure 6.6: Execution time

Although they did not look at the effect of Hyper-threading by itself, they looked at the effect of pinning processes to specific cores, where one strategy did not make use of Hyper-threading. They found that pinning the processes to different cores reduces energy consumption and variability. They attribute this to the reduced context switching, as spreading the processes while using Hyper-threading to pin two processes to one core showed increased consumption and variability. Of note is that they disabled C-states, which reduced the energy consumption variability but increased overall consumption. C-states adapt the frequency of the CPU to the workload, by disabling this functionality, the CPU runs at a higher frequency. Changing the governor of the CPU to performance is comparable, as it increases the default frequency of the CPU on the Lenovo P1 from 800MHz to 4Ghz.

In the second measurement results, a similar result can be observed as by Ournani et al. Disabling Hyper-threading reduces the variability while setting the governor also reduces the variability. Combining these settings further reduces variability. Although it was not tracked how the processes were spread across the cores, this could be one explanation for why disabling the Hyper-threading showed an increased energy consumption in the initial measurement. If the benchmark made use of more than 6 threads, disabling Hyper-threading would lead to extra context switching. In the second experiment, the new compilation might have adapted to the reduced number of cores available, thus removing this additional context switching.

The work by Ournani et al. was published while this research was taken place. Their research was focused on the hardware settings, while this research main focus was compiler settings. However, it is interesting to see that the limited hardware

settings that were measured show results that match their findings.

6.4 Validity

The measurements for the impact of the hardware settings were performed using the Ubuntu operating system. It is possible that using a different Linux distribution, a Windows distribution or an iOS installation could produce different results. The use of Ubuntu as an operating system also has another impact, as it means the software is executed using .NET Core. On a Windows installation, a different .NET version could be used which might alter the results. However, the drivers required to perform these tests on Windows are currently not available.

It is also possible that dust build-up in the fans of the used hardware affected the results of the tests. Similarly, the position of the system under test or the room temperature could influence the results.

Finally, software that puts a different type of load on the system than the benchmarks used in our measurements could show different results. Such loads might be affected by the hardware settings. This risk was already reduced by the fact that we used multiple different benchmarks, however, these benchmarks were mainly developed to compare programming languages and not to test the hardware performance.

6.5 Conclusion

Based on the results for the hardware settings measurements, we can provide an answer for RQ2: How do hardware settings influence the energy consumption of C# software?

We found that switching to performance mode increased energy consumption, which is what is to be expected. We also found that disabling hyper-threading can reduce energy consumption. These findings are in line with other research.

Interestingly, we obtained different results if we recompiled the code after changing hardware settings compared to only changing hardware settings. We hypothesise that the C# software fails to adapt to the reduced number of cores available when disabling hyper-threading without recompiling. This can explain the observed increase in energy consumption, as this would require additional context switching.

When changes are made to reduce energy consumption, it might be useful to test that the changes have the desired impact.

Chapter 7

Compiler Settings

In this chapter, we discuss the impact of different C# compiler settings on the energy consumption of the software. We start by analysing one of the settings that we tested to verify if it behaves as expected in Section 7.1. We then discuss the results of performing the benchmark tests with different compiler settings in Section 7.2. A statistical analysis of these results is presented in Section 7.3. Finally, we give an answer to RQ3 in Section 7.5.

7.1 QLRT

As explained in Section 3.4, .NET offers a number of compiler settings, namely Quick JIT (Q), Quick JIT for loops (L), ReadyToRun (R) and Tiered Compilation (T). While C# can be compiled using the `build` command, ReadyToRelease requires the `publish` which is used to publish an application and its dependencies for deployment. It also requires information about the platform that the application will be deployed on. According to the documentation, Tiered compilation will start with the code compiled by Quick JIT or ReadyToRelease. As the different compilation commands place the compiled files in different locations, it was not clear what happens when Quick JIT and ReadyToRelease are both enabled in the compilation settings.

To test what occurs when all settings are enabled (QLRT), we used both compilation methods and executed the code from both locations. We then compared the results to the results of using the comparable compilation, QLT and RT. The energy consumption results can be found in Figure 7.1 and the execution time can be found in Figure 7.2. It can be seen that the energy consumption and execution time of QLRT while executing the published code is equal to the energy consumption and execution time of RT. Similarly, the execution time and execution time of QLRT while executing the build code is equal to QLT. From this, it can be concluded that Quick JIT and ReadyToRelease should be treated as exclusive options and it is not possi-

ble to use both. As QLRT is thus not a real possibility, it will be excluded from the other results.

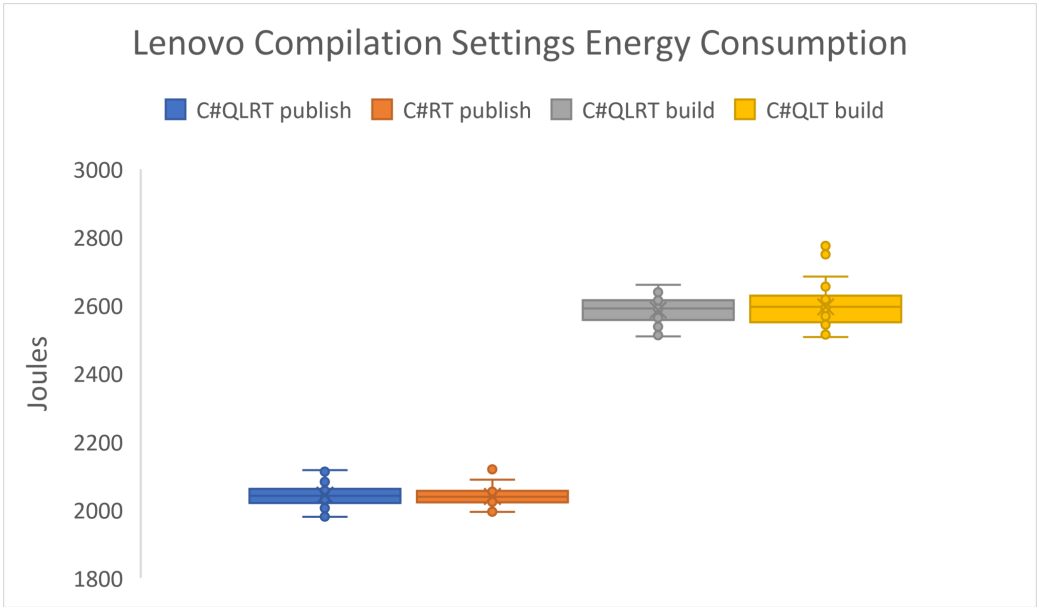

Figure 7.1: Energy consumption comparison of QLRT setting.

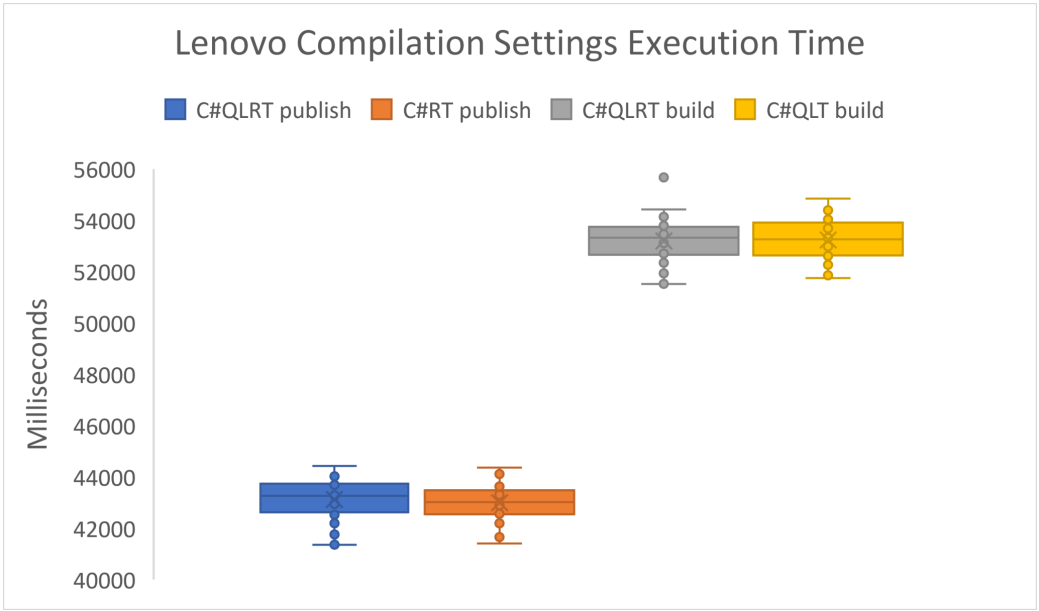

Figure 7.2: Execution time comparison of QLRT setting.

7.2 Compiler setting comparison

The measurements for the different compiler settings were executed with Hyper-threading enabled and with Hyper-threading disabled. The results of the measurements with Hyper-threading enabled can be seen in Figure 7.3 and Figure 7.4. The results of the measurement with Hyper-threading disabled can be seen in Figure 7.5 and Figure 7.6.

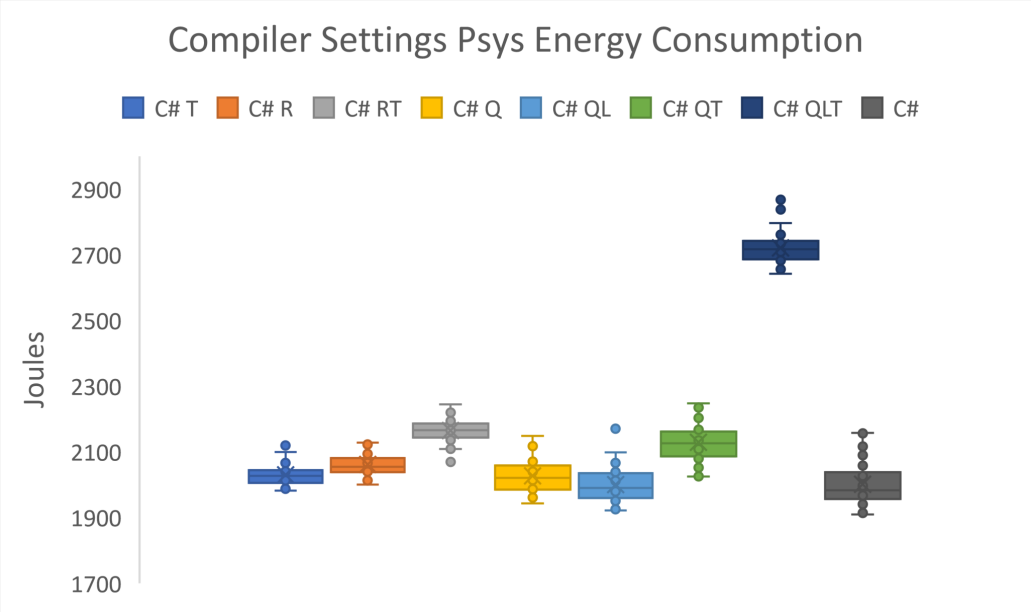

Figure 7.3: Energy consumption of different compiler settings.

The first thing that can be seen from the results of the measurements that one combination of settings performs considerably worse on both energy consumption and execution time. This is the combination of Quick JIT, Quick JIT for loops and Tiered compilation. This is an unexpected finding, as Tiered compilation is supposed to optimise frequently used code. The shorter compile time of Quick JIT is supposed to save more than the cost of running sub-optimal code before Tiered compilation can perform optimisations.[1] Overall, combining a setting with Tiered compilation seems to increase the energy consumption and execution time, but this impact is limited for RT and QT compared to QLT.

When looking at the measurement results for QLT in more detail, the increase in energy consumption is not the same for every benchmark. Spectral-norm is impacted the strongest, tripling the energy consumption and execution time, followed by Fannkuch-redux with a doubled energy consumption and execution time. In com-

[1] https://github.com/dotnet/runtime/blob/master/docs/design/features/tiered-compilation.md

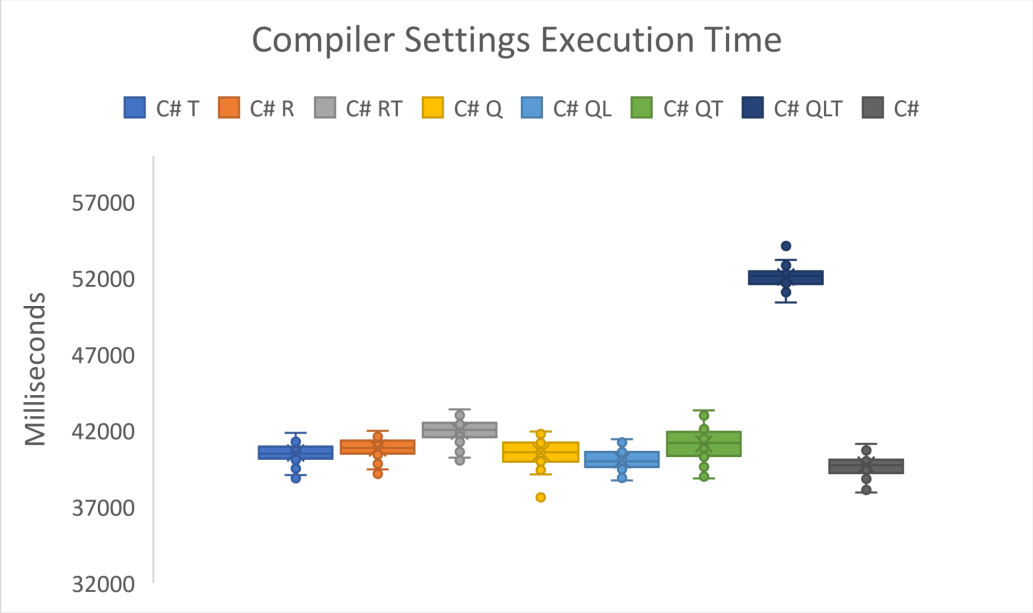

Figure 7.4: Execution time of different compiler settings.

parison, Pidigits is barely affected. The average energy consumption for the benchmarks can be found in Table 7.1.

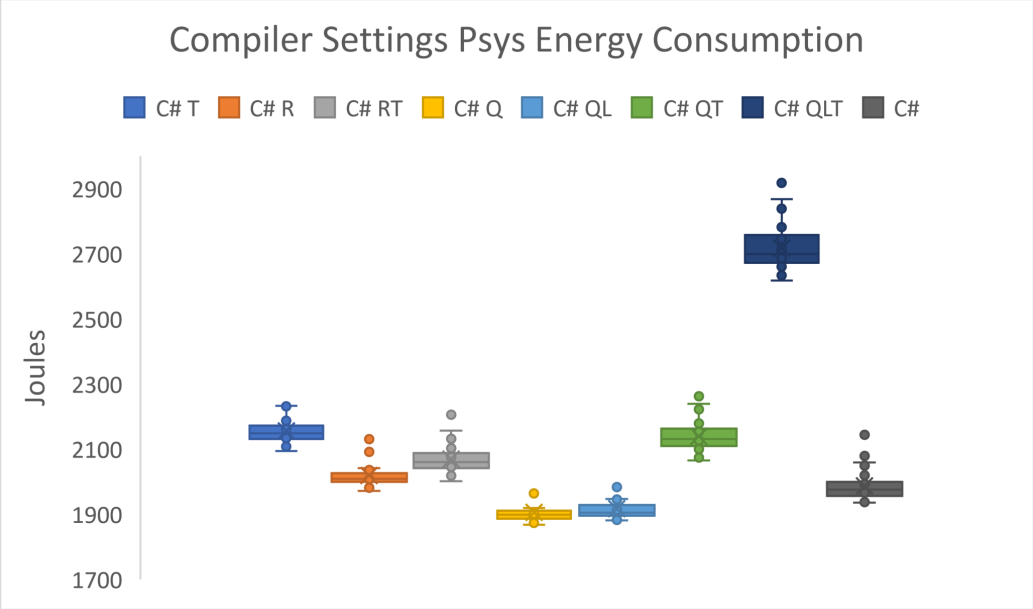

Figure 7.5: Energy consumption of different compiler settings with Hyper-threading disabled.

A second observation is that ReadyToRelease seems to reduce the variability of the energy consumption measurements. Almost all settings appear to reduce the variability of the basic version that does not use any of the compilation settings.

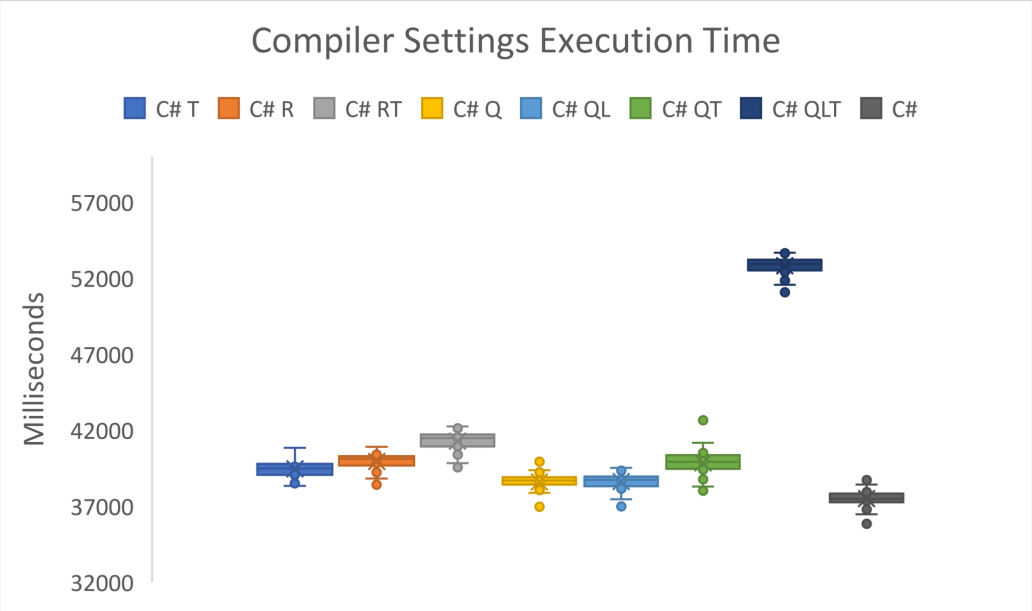

Figure 7.6: Execution time of different compiler settings with Hyper-threading disabled.

However, the total energy consumption for most options is very similar to the energy consumption when all settings are disabled. As of .NET Core 3.0, the default setting is QT. From these tests, removing Tiered compilation from the default settings could reduce energy consumption slightly. The execution time of Q and QT overlap, with QT having a larger variability.

By comparing Figure 7.3 and Figure 7.5, it is possible to see that the conclusion from Chapter 6, that Hyper-threading reduces the variability of the energy consumption also holds for these tests. However, this does not appear to be true for the energy consumption of QLT.

In Figure 7.7, the energy consumption is shown in Joules per millisecond. This has been calculated by dividing the total energy consumption by the execution time. In this figure, it can be seen that energy consumption varies slightly. However, the severely increased energy consumption of QLT can not be explained by the energy consumption per millisecond. This seems to indicate that when using QLT, the increased energy consumption is linked to the increased execution time. For all other settings, the relation between the total energy consumption and energy consumption per millisecond seems to match.

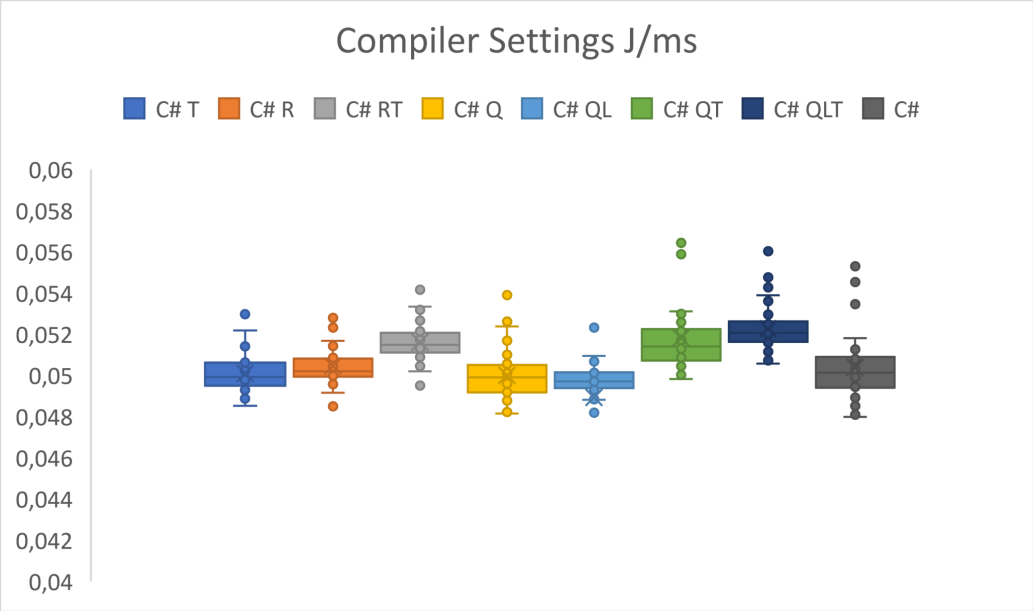

Figure 7.7: Energy consumption in Joule per millisecond of different compiler settings.

7.3 Statistical Analysis

As described in Section 4.3, we performed statistical analysis to verify that the differences observed in Section 7.2 are statistically significant. The average energy consumption of the benchmarks with different compilation settings can be found in Table 7.1. The results of the Wilcoxon ranksum test can be found in Table 7.2, Table 7.3, Table 7.4, Table 7.5 and Table 7.6. To reject the null hypothesis that two sample sets come from the same distribution with 95% confidence, the p-value should be smaller than 0.05. If the p-value is larger than 0.05, the null hypothesis can not be rejected. In the tables, white cells indicate that the null hypothesis can not be rejected. The results of one setting are compared to other settings, so in Table 7.2, the first column contains the results for comparing C# to C#T. Green and yellow cells indicate a significant difference, with the energy consumption of the column's setting lower than the setting it is compared to. Orange and red cells indicate that the column's setting has a higher energy consumption than the setting it is compared to. Red and green are used for very low p-values, indicating high confidence that they are significantly different. Yellow and orange indicate that the p-value is closer to the decision value.

From these results, it can be seen that QLT uses significantly more energy. However, it also becomes clear that for the other settings, it is harder to draw a clear conclusion. For example, when comparing C#Q to C#R, some benchmarks consume significantly more energy, while others consume significantly less energy. As

	C#T	C#R	C#RT	C#Q	C#QL	C#QT	C#QLT	C#
Binary	375,5	377,5	387,8	375,4	380,0	398,4	448,4	373,4
Fasta	65,2	63,8	64,8	66,4	65,4	67,2	71,8	67,8
Fannkuch-red.	387,9	446,4	453,3	422,7	409,4	404,1	859,4	413,7
K-nucleotide	215,5	251,6	248,9	230,2	226,0	220,4	250,1	225,8
N-body	128,3	131,4	131,0	134,8	134,2	138,7	136,6	132,5
Pidigits	30,0	29,2	30,0	30,9	30,4	31,3	30,3	30,1
Reverse-comp.	28,2	31,8	32,0	29,3	28,5	29,9	45,3	28,9
Spectral-norm	24,0	27,7	28,1	24,0	23,6	24,9	75,4	24,3
Regex-redux	656,3	581,1	674,3	586,1	578,4	686,9	671,6	579,8
Mandelbrot	119,0	120,5	123,5	127,0	123,1	127,4	131,5	125,2

Table 7.1: The average Psys energy consumption of different compiler settings. Reported values are in Joules, rounded to 1 decimal value

can be seen in Table 7.1, regex-redux, binary and fannkuch-redux consume most of the energy. When comparing two compilation settings, these benchmarks thus dominate the consumption trends.

To better understand when energy consumption is increased and when it is decreased, the code of the benchmarks should be analysed in more detail. There might be certain coding structures that show up in benchmarks that show similar energy consumption behaviour. We can already see that the difference cannot be explained based on whether a benchmark makes use of parallel programming. K-nucleotide and N-body both don't make use of parallel programming but often show different energy consumption, with one consuming more energy while the other consumes less.

	C# compared to:						
	C#T	C#R	C#RT	C#Q	C#QL	C#QT	C#QLT
Binary	0.16	0.01	0.004	0.25	0.001	<0.001	<0.001
Fasta	<0.001	<0.001	<0.001	0.013	<0.001	0.21	<0.001
Fannkuch-red.	0.15	0.004	<0.001	0.032	0.0018	0.22	<0.001
K-nucleotide	<0.001	<0.001	<0.001	0.011	0.47	0.0054	<0.001
N-body	<0.001	0.012	0.002	<0.001	0.002	<0.001	<0.001
Pidigits	0.20	<0.001	0.28	<0.001	0.016	<0.001	0.11
Reverse-comp.	<0.001	<0.001	<0.001	0.009	0.0026	<0.001	<0.001
Spectral-norm	<0.001	<0.001	<0.001	0.06	<0.001	<0.001	<0.001
Regex-redux	<0.001	0.30	<0.001	0.031	0.47	<0.001	<0.001
Mandelbrot	<0.001	<0.001	0.016	0.002	<0.001	<0.001	<0.001

Table 7.2: P-values of Wilcoxon rank-sum test comparisons of different compiler settings vs C#. Green indicates a significantly lower energy consumption and orange and red indicate a significantly higher energy consumption.

	C#Q compared to:					
	C#T	C#R	C#RT	C#QL	C#QT	C#QLT
Binary	0.39	0.072	0.0031	0.015	<0.001	<0.001
Fasta	0.065	<0.001	0.0046	0.046	0.044	<0.001
Fannkuch-redux	<0.001	0.16	<0.001	<0.001	0.022	<0.001
K-nucleotide	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
N-body	<0.001	<0.001	<0.001	0.14	<0.001	0.001
Pidigits	<0.001	<0.001	<0.001	0.002	0.006	0.0036
Reverse-comp.	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Spectral-norm	0.056	<0.001	<0.001	0.0013	<0.001	<0.001
Regex-redux	<0.001	0.11	<0.001	0.0012	<0.001	<0.001
Mandelbrot	<0.001	<0.001	<0.001	<0.001	0.28	<0.001

Table 7.3: P-values of Wilcoxon rank-sum test comparisons of different compiler settings vs C#Q. Green indicates a significantly lower energy consumption and orange and red indicate a significantly higher energy consumption.

	C#QL compared to:				
	C#T	C#R	C#RT	C#QT	C#QLT
Binary	0.039	0.16	0.32	<0.001	<0.001
Fasta	0.43	0.01	0.23	<0.001	<0.001
Fannkuch-redux	0.24	<0.001	<0.001	0.004	<0.001
K-nucleotide	<0.001	<0.001	<0.001	0.0072	<0.001
N-body	<0.001	<0.001	<0.001	<0.001	0.001
Pidigits	0.0035	<0.001	0.0092	<0.001	0.19
Reverse-comp.	0.029	<0.001	<0.001	<0.001	<0.001
Spectral-norm	0.009	<0.001	<0.001	<0.001	<0.001
Regex-redux	<0.001	0.19	<0.001	<0.001	<0.001
Mandelbrot	<0.001	<0.001	0.18	<0.001	<0.001

Table 7.4: P-values of Wilcoxon rank-sum test comparisons of different compiler settings vs C#QL. Green indicates a significantly lower energy consumption and orange and red indicate a significantly higher energy consumption.

	C#QT compared to:					C#T to:
	C#T	C#R	C#RT	C#QLT		C#QLT
Binary	<0.001	<0.001	<0.001	<0.001		<0.001
Fasta	<0.001	<0.001	<0.001	<0.001		<0.001
Fannkuch-redux	<0.001	<0.001	<0.001	<0.001		<0.001
K-nucleotide	0.013	<0.001	<0.001	<0.001		<0.001
N-body	<0.001	<0.001	<0.001	0.0018		<0.001
Pidigits	<0.001	<0.001	<0.001	<0.001		0.0036
Reverse-comp.	<0.001	<0.001	<0.001	<0.001		<0.001
Spectral-norm	<0.001	<0.001	<0.001	<0.001		<0.001
Regex-redux	<0.001	<0.001	0.0097	0.0016		<0.001
Mandelbrot	<0.001	<0.001	<0.001	<0.001		<0.001

Table 7.5: P-values of Wilcoxon rank-sum test comparisons of different compiler settings vs C#QT and vs C#T. Green indicates a significantly lower energy consumption and orange and red indicate a significantly higher energy consumption.

	C#R compared to:			C#RT compared to:	
	C#T	C#RT	C#QLT	C#T	C#QLT
Binary	0.14	0.33	<0.001	0.067	<0.001
Fasta	0.006	0.031	<0.001	0.14	<0.001
Fannkuch-redux	<0.001	<0.001	<0.001	<0.001	<0.001
K-nucleotide	<0.001	0.12	0.33	<0.001	0.23
N-body	<0.001	0.23	<0.001	<0.001	<0.001
Pidigits	<0.001	<0.001	<0.001	0.35	0.042
Reverse-comp.	<0.001	0.12	<0.001	<0.001	<0.001
Spectral-norm	<0.001	0.004	<0.001	<0.001	<0.001
Regex-redux	<0.001	<0.001	0.0016	<0.001	0.30
Mandelbrot	< 0.011	0.0019	<0.001	<0.001	<0.001

Table 7.6: P-values of Wilcoxon rank-sum test comparisons of different compiler settings vs C#R and C#RT. Green indicates a significantly lower energy consumption and orange and red indicate a significantly higher energy consumption.

7.4 Validity

The validity of the observed differences in energy consumption for different compiler settings could be affected by several factors. Firstly, similarly to what was mentioned in Section 6.4, the type of operating system used to perform the tests could influence the results. This is especially relevant since C# and .NET Framework were originally developed for Windows. The tested compiler options are only offered by .NET Core, which was developed for cross-platform development. This does mean that these results should not be compared to software developed with other .NET versions. However, it is possible that different versions of .NET Core could produce different results.

It is also possible that executing energy consumption measurements with different software programs could produce different results. We already noted that for some compiler settings, certain benchmarks showed a reduction in energy consumption while other settings showed an increased energy consumption. As such, larger programs with longer execution times could produce different results.

Longer or more complex programs would likely most affect the Tiered compilation setting. According to the official description, this setting attempts to optimise the initial compilation provided by Quick JIT or ReadyToRelease during the execution. As such, if a program executes code sections multiple times, it increases the chances for optimisations to be found and for the optimisations to show their effect.

7.5 Conclusion

Based on our findings, we can provide an answer to RQ3: How do C# compiler settings influence the energy consumption of software?

We found that there is one compilation setting that should be avoided, namely QuickJIT combined with QuickJIT for loops and Tiered compilation, as it significantly increases energy consumption and execution time. We further found that one combination of settings (QLRT) should not be used as Tiered compilation uses either the Quick JIT code or the ReadyToRelease code based on what files are used to start the program. For the other settings, we found out that there are significant differences between the compiler settings, but that the impact differs between benchmarks. Further research is necessary to discover when the energy consumption is reduced and when it is increased before general guidelines can be given. However, as there are significant differences, we can advise performing tests to find which settings reduce the energy consumption of your software.

We observed that when comparing compiler settings, it is possible for some benchmarks to show reduced energy consumption while other benchmarks show

an increased energy consumption. This could indicate that the performance of compiler settings is influenced by the programming structures that are used. Further research is required to verify if there is such a correlation.

Chapter 8

Architectures

In this chapter, the results of executing the compiler setting tests on different hardware architectures are discussed. These tests were performed to investigate if the observations discussed in Chapter 7 were unique to the system under test or if they were applicable to a broader set of hardware.

8.1 Operating System

Since the tests on the different architectures are executed using a live USB boot version of Ubuntu, we also performed the tests on the original system using a live USB boot. By comparing the results of these tests to the results obtained from an installed Ubuntu version, we can see how the use of a live USB boot impacts the measurements.

The energy consumption measured on the live USB boot can be found in Figure 8.1, while Figure 8.2 shows the results of the installed boot with the same axis scale. Figure 8.3 and Figure 8.4 show the execution time of the live USB boot and installed boot respectively.

When comparing the charts, it can be seen that the trends stay the same, but the energy consumption on the live USB boot is approximately 500 Joules higher. The execution time is also increased by approximately 3000 milliseconds. Furthermore, when looking at the energy consumption per time unit, as shown in Figure 8.5, it can be seen that the energy consumption per millisecond is also increased. Interestingly, the variability of the energy consumption is reduced. The increased energy consumption on the live USB boot can be explained by the fact that it is limited to using the CPU and RAM. The hard disk storage can not be used, thus increasing the CPU activity.

Since the trends in energy consumption are equal, we conclude that live USB boot can be used to perform tests on other architectures. The reported values are

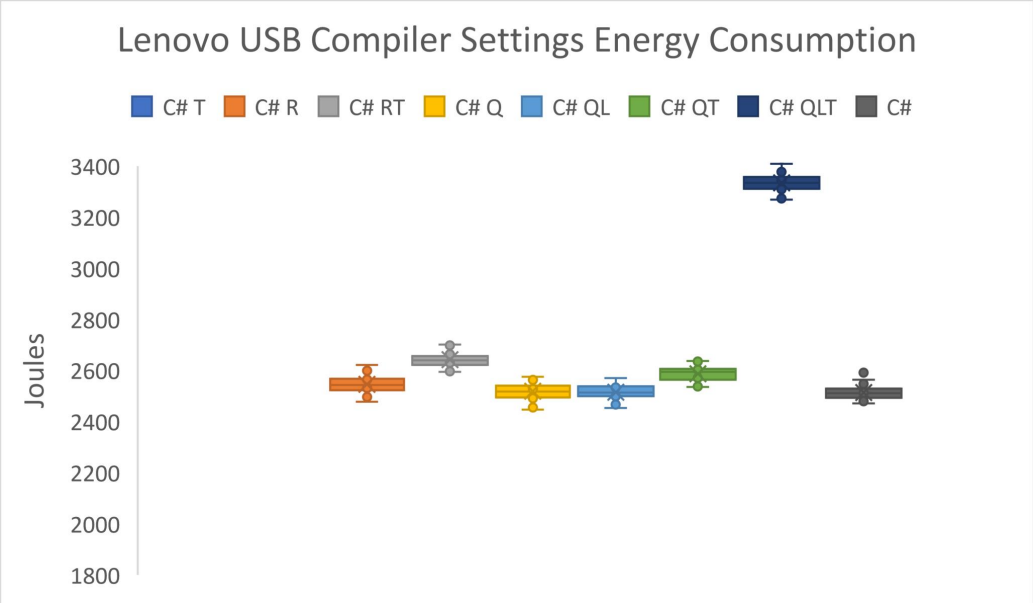

Figure 8.1: Psys energy consumption of different compiler settings on Lenovo P1 with USB boot.

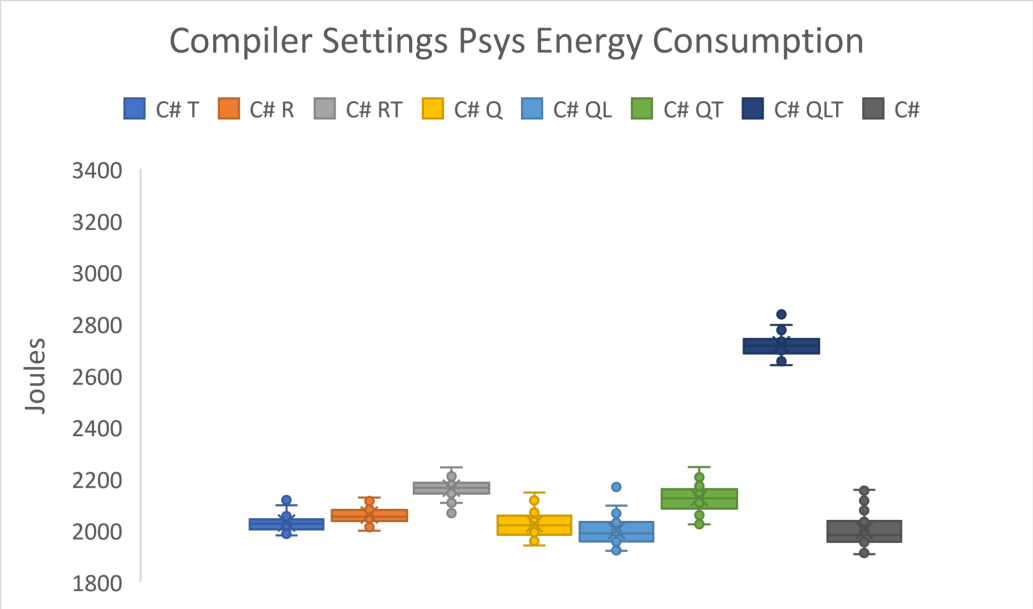

Figure 8.2: Psys energy consumption of different compiler settings on Lenovo P1 with normal boot.

higher then they would be on an installed boot, but the comparison of different compilation settings remains valid.

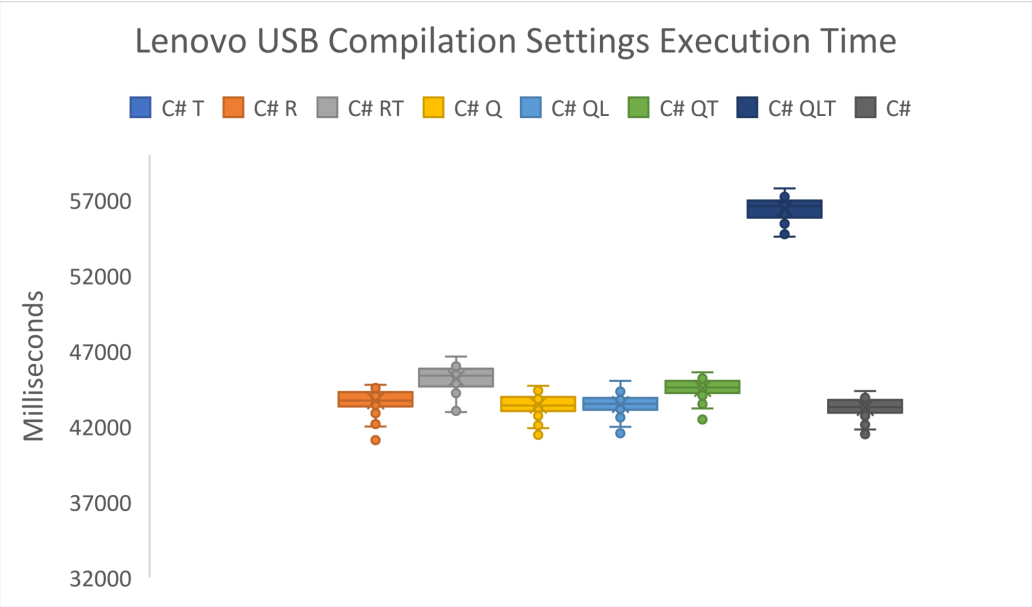

Figure 8.3: Execution time of different compiler settings on Lenovo P1 with USB boot.

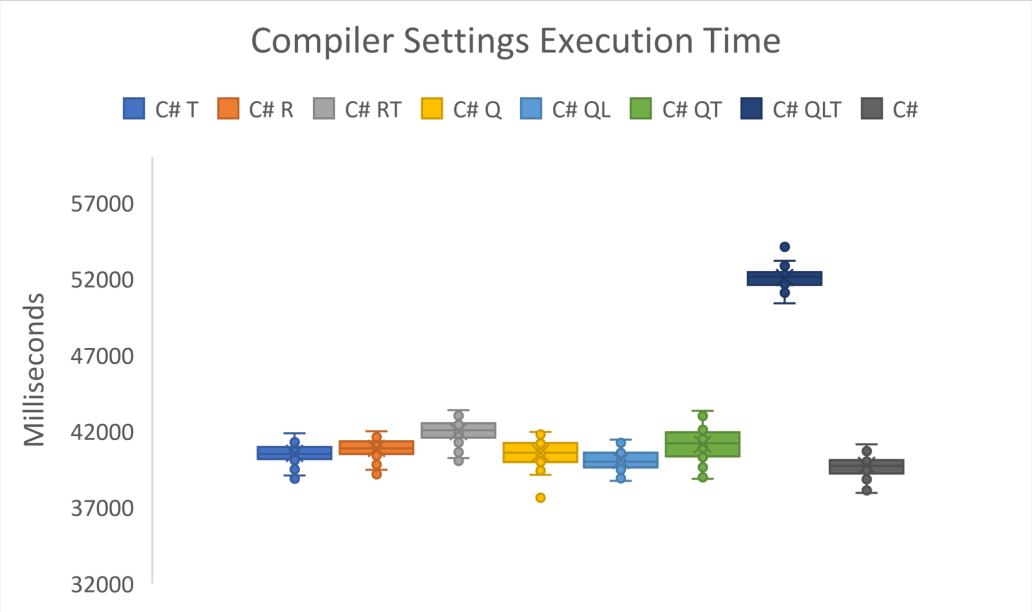

Figure 8.4: Execution time of different compiler settings on Lenovo P1 with normal boot.

8.2 Results

The tests were executed on a 4th generation i7 CPU and two 6th generation i5 CPUs. The 4th generation CPU does not support the Psys domain, as this had not yet been introduced by Intel. The Psys domain was introduced with the 6th

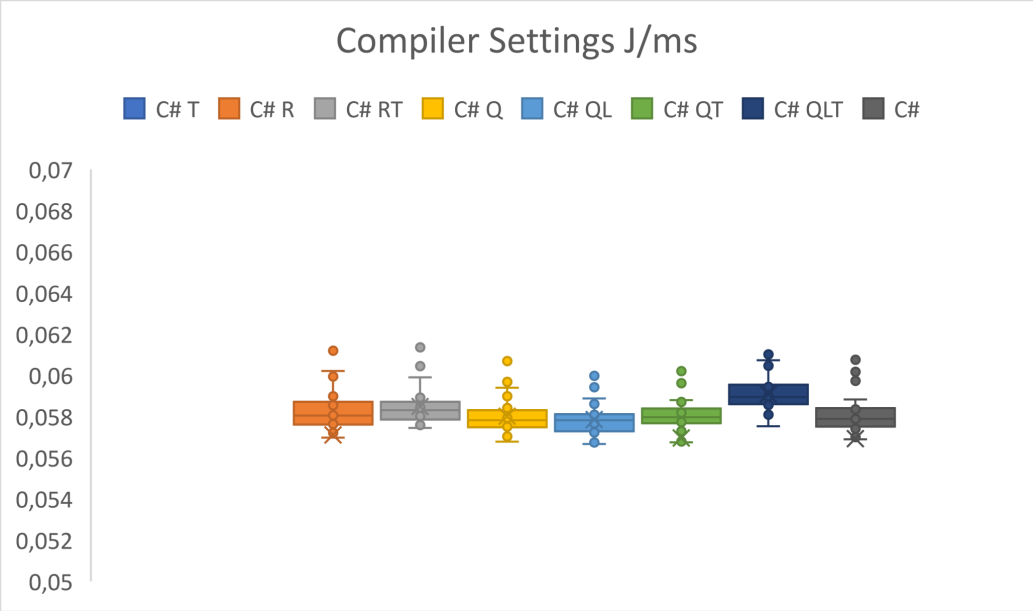

Figure 8.5: Energy consumption in Joule per millisecond of different compiler settings on Lenovo P1 with USB boot.

generation (Skylake), however, not all CPU versions support this domain. This is also the case with one of the 6th generations CPUs used to perform tests. It is possible to read data from the Psys domain, however, the data is incorrect. The Psys domain is supposed to cover all on-chip systems, but the values reported on this particular system are half or even a quarter of the values reported by the Package domain. As the Psys is supposed to encompass the Package domain, we concluded that it was not supported on this model. For the systems that do not support the Psys domain, the Package domain is used instead.

The results of executing the tests on the Dell Latitude, supporting the Psys domain, can be found in Figure 8.6 and Figure 8.7. The results of executing the tests on the Dell Precision and HP Elitebook, using the Package domain, can be found in Figure 8.9, Figure 8.10, Figure 8.11 and Figure 8.12. Since these systems use a different domain, the results of executing the tests on the Lenovo are displayed in Figure 8.8 based on the Package domain instead of the Psys domain.

It is important to note that on the Dell Latitude, the Tiered compilation setting with all other settings disabled was not executed, due to the limited availability of the system. Furthermore, on the Dell Precision and HP Elitebook, the test with all settings disabled was not executed. They are included in the legends of the figures to improve comparability, but no data is displayed.

It is interesting to see that enabling Quick JIT, Quick JIT for loops and Tiered compilation (QLT) is the worst performing set of settings on all architectures. Furthermore, the other settings also show approximately the same pattern. The actual

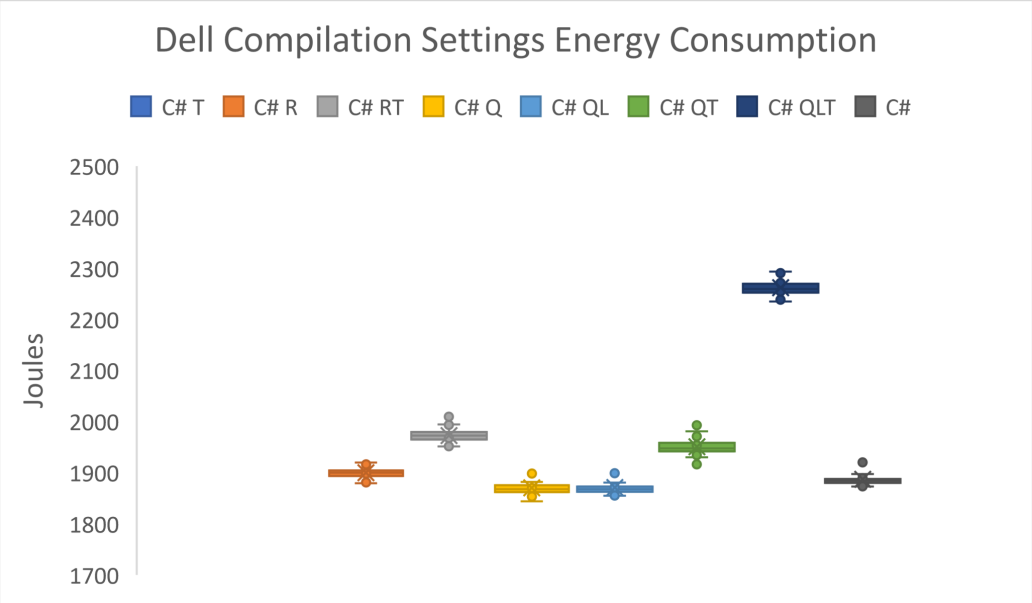

Figure 8.6: Psys energy consumption of different compiler settings on Dell Latitude

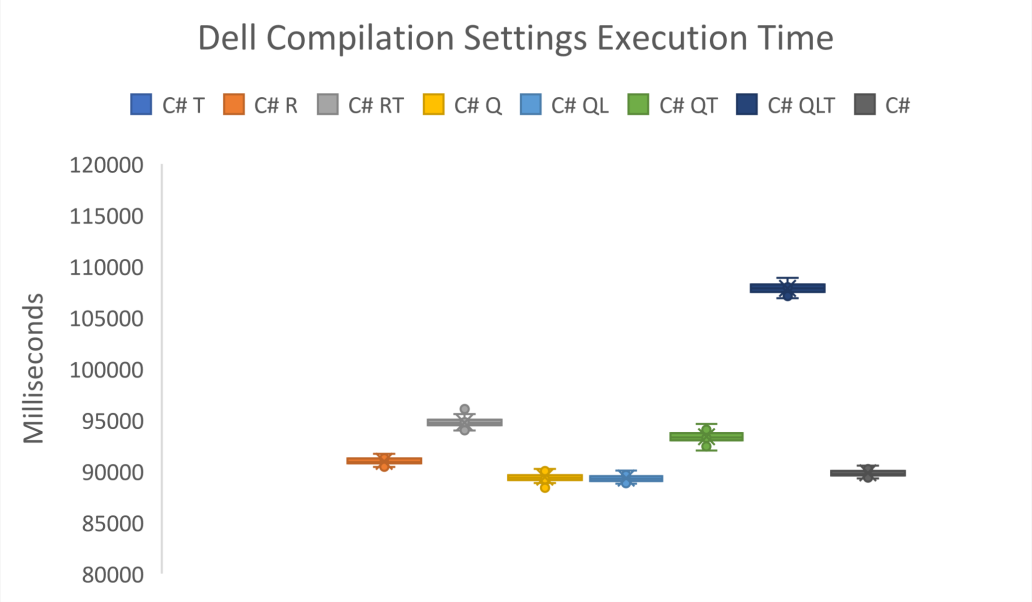

Figure 8.7: Execution time of different compiler settings on Dell Latitude

energy consumption and execution time differ between architectures, but the relative relations stay the same.

The tests performed on the other architectures also show the reduced variability that was discussed in Section 8.1. As explained, all actions and storage are handled entirely by the CPU and RAM. This removes possible impacts caused by retrieving data from disk compared to from memory, as everything is already available in mem-

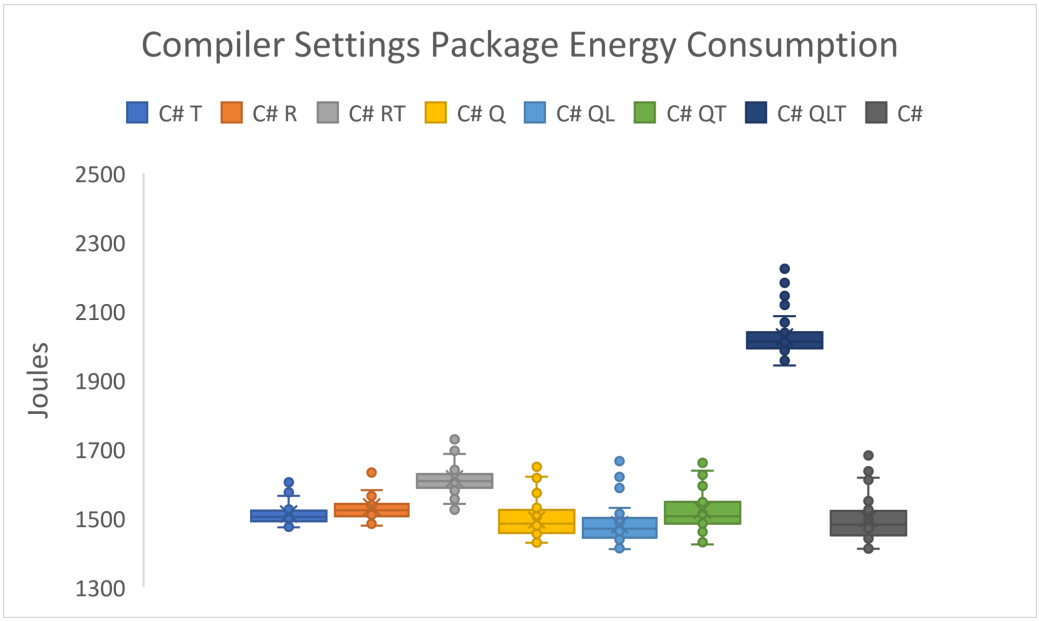

Figure 8.8: Package energy consumption of different compiler settings on Lenovo P1

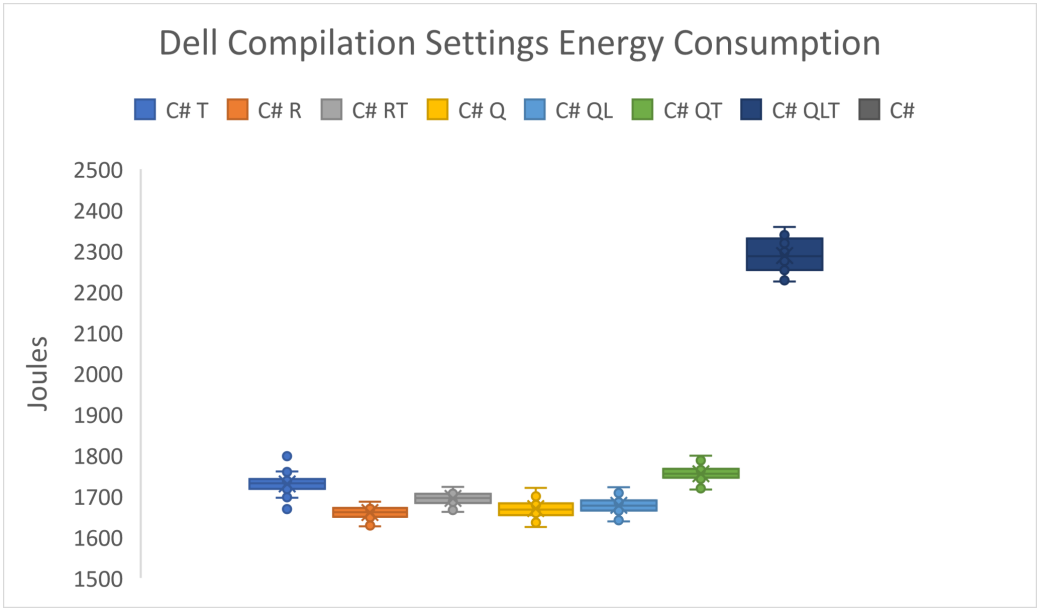

Figure 8.9: Package energy consumption of different compiler settings on Dell Precision

ory. Furthermore, it is possible that the number of background processes is reduced in a live USB boot compared to an installed boot.

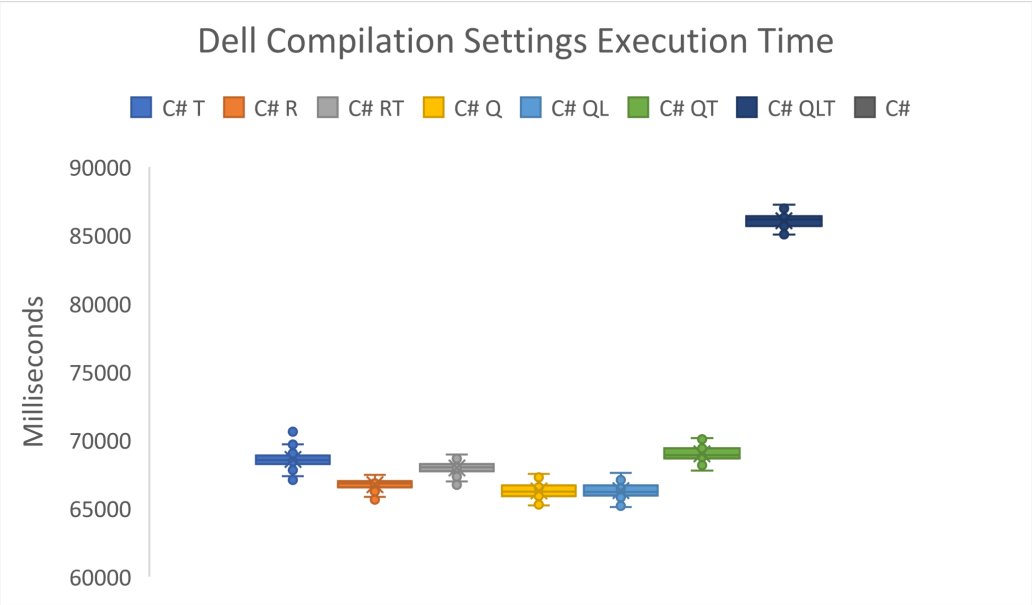

Figure 8.10: Execution time of different compiler settings on Dell Precision

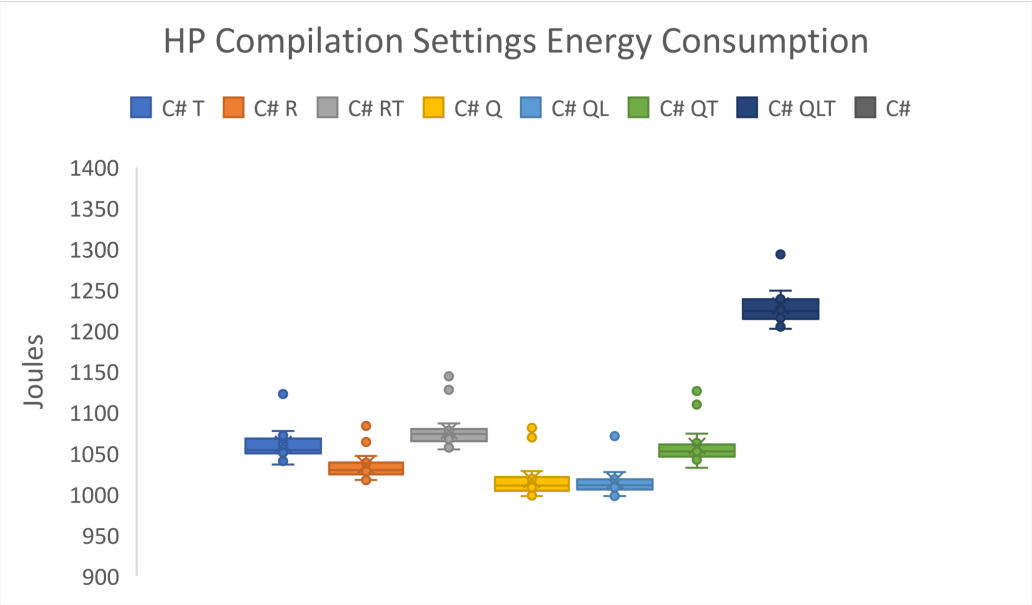

Figure 8.11: Package energy consumption of different compiler settings on HP Elitebook

8.3 Validity

The validity of the results presented in this chapter can be affected by several factors. First of all, the tests were run without installing the operating system, instead, they were performed with the use of a live USB boot. This risk was mitigated by also performing the tests with the live USB boot on the original system. This showed that

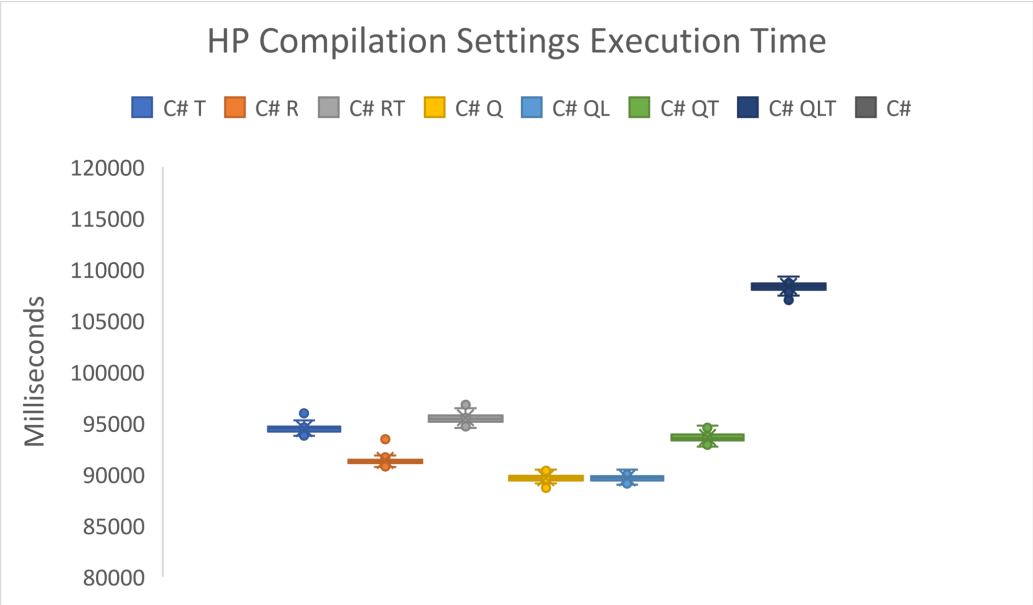

Figure 8.12: Execution time of different compiler settings on HP Elitebook

both energy consumption and execution time increased by using a live USB boot, but the trends remained the same. It is likely, but not guaranteed that this is also the case on the other hardware systems.

Secondly, research has shown that the accuracy of the RAPL energy consumption measurements has increased with newer versions. This also indicates that the results on the older systems could be less accurate. Since we intended to compare the overall trends and not the precise energy consumption, the impact of the lower accuracy is reduced.

Finally, all tested systems were laptops with CPUs for laptops. It is possible that CPUs for servers could produce different results. Furthermore, the tested systems are all Intel CPUs. CPUs from different manufacturers could also provide different results, but energy consumption measurements can not be performed using Intel's RAPL on such CPUs.

8.4 Conclusion

Based on the presented results, we can provide an answer for RQ4: How consistent is the impact of compiler settings across hardware architectures?

These tests indicate that the compiler settings offered by .NET show a consistent energy consumption on different architectures. It is thus likely that if a setting is found to reduce the energy consumption of software on one architecture, this will also be true for other systems.

Chapter 9

Programming Choices

In this chapter, we propose programming choices that could be measured to discover if they impact energy consumption. We discuss why the choices might produce differences in energy consumption as well as examples of code that can be used to perform experiments.

9.1 Loops

In their best practices for performance on Android, Google suggests using the enhanced for loop, or for-each loop, for collections with an iterator. They also mention that a hand-written counted loop is three times faster for an ArrayList. As mentioned in Chapter 2, Tonini et al. [26] analysed the energy consumption of for loops. They found that the hand-written counted loop was not only faster but also reduced energy consumption. Furthermore, they found that providing a loop with the collection length reduces the energy consumption compared to requesting the length at every iteration.

In C#, the size of a collection is an attribute of the collection. It is thus not a method call but a property access. The only exception is the GetLength method that can retrieve the length of an array based on a given dimension. Arrays also have a length property, this method can be used to obtain the length of a specific dimension in a multidimensional array. As a property access instead of a method call, it is possible that there is a smaller impact on energy consumption or even no detectable impact.

C# also has an enhanced for loop, which can be used on instances of types that implement the IEnumerable interface or types that provide a GetEnumerator method with the correct return type. Furthermore, the type of the variable being iterated can be explicitly defined or left for the compiler to infer. The type inference needs to check if the type is iterable, which could increase energy consumption.

There could be a difference in energy consumption between an enhanced for loop and a normal for loop. In Listing 1, we provide the code for four methods that implement the same functionality but use the different syntax possibilities discussed. The collection used in the methods could be changed to detect if this has an impact on the energy consumption results.

```
Public Class ForTest
{
    public void One()
    {
        int numbers = new List<int> {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        int sum = 0;

        for(int i = 0; i < numbers.Count; i++)
        {
            sum+= List[i];
        }
    }

    public void Two()
    {
        int numbers = new List<int> {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        int sum = 0;
        int len = numbers.Count;

        for(int i = 0; i < len; i++)
        {
            sum+= List[i];
        }
    }

    public void Three()
    {
        int numbers = new List<int> {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        int sum = 0;

        foreach(int number in numbers)
        {
            sum+= number;
        }
```

```
    }

    public void Four()
    {
        int numbers = new List<int> {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        int sum = 0;

        foreach(var number in numbers)
        {
            sum+= number;
        }
    }
}
```

Listing 1: Methods implementing the same functionality with different for loop syntax.

9.2 Pattern Matching

The release of C# 8.0 expanded the possibilities for pattern matching. At the 2020 NDC conference in London, Bill Wagner[1] showed how the new capabilities could be used to rewrite a sample program that calculates a toll rate based on a number of parameters. The initial code used a number of conditional statements to check the parameters. By grouping the parameters into a tuple and making use of the new switch expression, the lines of code were significantly reduced. Using wildcards then further reduced the number of conditions. This reduced the original 50 lines of code to 11 lines. While the functionality remained the same, this makes it much easier to see what the function does. However, since it is a completely different syntax, it is unclear how this is handled by the compiler. Furthermore, it is not yet known if or how this impacts performance and energy consumption. In Listing 2 and Listing 3 we show the relevant portions of the code. The complete code is provided in Appendix C

```
public static decimal PeakTimePremiumImperative
                    (DateTime timeOfToll, boolean inbound)
{
    if (IsWeekDay(timeOfToll))
    {
```

[1] Change your habits: Modern techniques for modern C# - Bill Wagner

```
if(inbound)
{
        var timeBand = GetTimeBand(timeOfToll);
        if (timeBand == TimeBand.MorningRush)
        {
            return 2.00m;
        }
        else if (timeBand == TimeBand.Daytime)
        {
            return 1.50m;
        }
        else if (timeBand == TimeBand.EveningRush)
        {
            return 1.00m;
        }
        else
        {
            return 0.75m;
        }
}
else
{
    var timeBand = GetTimeBand(timeOfToll);
    if (timeBand == TimeBand.MorningRush)
    {
        return 1.00m;
    }
    else if (timeBand == TimeBand.Daytime)
    {
        return 1.50m;
    }
    else if (timeBand == TimeBand.EveningRush)
    {
        return 2.00m;
    }
    else
    {
        return 0.75m;
    }
```

```
        }
    }
    else
    {
            return 1.00m
    }
}
```

Listing 2: Code for calculating a toll rate based on a set of parameters using conditional statements.

```
public static decimal PeakTimePremium
                (DateTime timeOfToll, boolean inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush,    true)   => 2.00m,
        (true, TimeBand.MorningRush,    false)  => 1.00m,
        (true, TimeBand.Daytime,        _)      => 1.50m,
        (true, TimeBand.EveningRush,    true)   => 1.00m,
        (true, TimeBand.EveningRush,    false)  => 2.00m,
        (true, TimeBand.Overnight,      _)      => 0.75m,
        (false, _,                      _)      => 1.00m,
    };
```

Listing 3: Code for calculating a toll rate based on a set of parameters using pattern matching.

9.3 LINQ

Language Integrated Query (LINQ) was introduced in .NET framework 3.5. It extends C# with query capabilities. Traditionally, queries were strings without type checking and with language difference for each data source. LINQ integrates queries into the C# language, making it a first-class member of the language, just like classes and methods. It can be used with strongly typed collections, providing a consistent method for interactions with objects, relational databases and XML. It allows grouping, filtering and ordering of data with the same pattern for interacting with different data sources.

To use LINQ to perform operations with data, three steps are necessary. First, the data has to be made available, either by loading it into memory or by creating a

link to an external data source such as an SQL database. The second step is the definition of the query and storing it in a query variable, while the third and final step executes the query using a foreach loop. Queries that perform an aggregation over a range of elements require an iteration over these elements and do not require an explicit foreach loop.

LINQ can be used to create smaller code that can be easier to understand and maintain. However, it is a generic approach to solving problems. This inherently comes with additional overhead. Furthermore, when you search "LINQ performance", a lot of results come up explaining how LINQ code is slower than manually written for loops. In some cases, the performance can be attributed to inefficient use of query methods. For example, if data should be selected and ordered, it is more efficient to first select the part of the data you want ordered, instead of starting by ordering the data and then selecting the part of the data you are actually interested in. The introduction of .NET Core also improved the performance of LINQ queries.

It is possible that the differences in performance are linked with a difference in energy consumption, or that the energy consumption shows different trends. Section 9.3 provides some small code that could be used to test the difference between LINQ queries and an iterative approach.[2] Section 9.3 shows two functionally equivalent queries, but due to the order of the queries, the performance can differ. The difference in performance is based on how much of the data is excluded in the selection.[3]

```
public int Iterative()
    {
        var counter = 0;
        foreach (var item in items)
        {
            if (item % 10 == 0)
                counter += item + 5;
        }

        return counter;
    }

public int Linq()
    {
        var results = items.Where(i => i % 10 == 0).Select(i => i + 5);
        var counter = 0;
```

[2]Source: GitHub: LinqOptimisation-WhereSelectBenchmark
[3]Source: Stackoverflow: Does the order of LINQ functions matter?

```
        foreach (var result in results)
        {
            counter += result;
        }

        return counter;
    }
```

Listing 4: LINQ and iterative code selecting and manipulating a sequence of numbers.

```
var query = myCollection.OrderBy(item => item.CreatedDate)
                        .Where(item => item.Code > 3);

var query = myCollection.Where(item => item.Code > 3)
                        .OrderBy(item => item.CreatedDate);
```

Listing 5: Two LINQ queries with a different order of the queries.

9.4 Validity

If measurements are performed with the code suggested in this chapter, several factors could influence the results. The duration of the code under test should be considered, as the overhead costs and timing of RAPL register updates become important factors for short code paths. This could be mitigated by repeating the code under test multiple times in one measurement, increasing the execution time.

Furthermore, if a function is tested where parameters decide the execution path, the choice of parameters could influence the results. This can be mitigated by performing tests with different paths to ensure that all execution paths are measured.

Finally, a different framework version could produce different results. Developments in newer versions could change the behaviour of programming structures, improving the performance and changing the energy consumption.

9.5 Conclusion

In this chapter, we suggest experiments that can be performed in order to provide an answer for RQ5: How do functionally equal programming choices impact energy consumption?

As we did not perform the experiments, it is not possible to make a conclusion. However, we did encounter anecdotal evidence that there are performance differences when comparing LINQ queries to imperative approaches. The analysis of the energy consumption measurements for different compiler settings showed that increased energy consumption was probably caused by the increased execution time. This also suggests that differences in performance will be linked to differences in energy consumption.

Chapter 10

Conclusions & Recommendations

In Section 10.1, we summarise the threats to the validity of our research. In Section 10.2, we summarise the conclusions and in Section 10.3 we suggest possible directions for further research

10.1 Validity

A number of issues can affect the validity of our work. Energy measurements were performed using the Intel RAPL tool, which has evolved with the CPU generations to become known as one of the most accurate tools for CPUs. However, when sampling at a high frequency, the variability in the timing of the updates can cause problems. Furthermore, sampling at a high frequency introduces additional overhead costs. The tests performed in this research were of sufficient length that these issues could be avoided.

The operating system adds a layer of uncertainty to performing measurements of software, as it is not clear how tasks are scheduled. This also means that it is unclear if the system will handle a test in the same manner if it is executed multiple times. This adds an inherent variability to the execution of software. We attempted to reduce the impact of this variability by repeating the tests multiple times.

The generalisability of the results could be impacted by the fact that all systems were notebooks. Furthermore, the tests were all executed under Ubuntu. Executing the tests under a different operating system, such as Windows, could produce different results. This could be especially true for C#, as the initial focus of C# and .NET was the Windows operating system. At this time, the drivers required to perform similar tests under Windows are not available.

Another factor is the version of the .NET framework used to compile the benchmark used in this research. Different versions of the framework could produce different results. As the compiler settings that were treated in this research are a feature

of .NET Core, it is not possible to make a comparison to different frameworks such as .NET. However, newer versions of .NET Core could show different results.

Finally, the experiments were performed with a set of benchmarks initially developed to compare the performance of different programming languages. These benchmarks are relatively small. Bigger, more complex programs could produce different results. This would most probably most affect the Tiered Compilation setting. According to its official description, this setting is meant to optimise code during execution. As such, programs that run for longer or that execute code segments multiple times in one run will probably benefit more from this setting.

10.2 Conclusions

Based on the information presented in this thesis, we can answer the research questions we started with. We first provide answers to the sub-questions before answering the main research question.

RQ1: How can developers obtain reliable results on the energy consumption of their software?

Intel's RAPL protocol can be used to perform energy measurements on Linux systems. There are multiple tools that can assist with these measurements. These make it easier to perform the measurements without requiring a lot of extra knowledge. As such tools can add overhead costs to the measurements, it would be preferred to at least understand how the tools work so you can be aware of such costs. Leaving a system idle, ensuring no other applications are active and that the system does not switch operation modes during the tests can provide reliable results. By repeating measurements, the impact of variability caused by the operating system can be reduced. It should be noted that executing tests under different circumstances can increase variability.

RQ2: How do hardware settings influence the energy consumption of software systems?

We analysed the impact of two hardware settings, namely the operating mode of the CPU governor and the use of Intel's Hyper-threading. We found that the performance governor increased the energy consumption compared to the power-save governor. Disabling the Hyper-threading setting, reduced the variability of the energy measurements, as well as reducing the execution time. Interestingly, in our initial tests, we obtained different results. These tests were performed without recompiling the code. When the tests were performed while recompiling the code after every hardware setting change, the Hyper-threading results changed. The Hyper-threading setting changes the number of available processors. If it is enabled, every physical core can act as two virtual cores. By disabling the Hyper-threading setting,

the number of available cores changes. One explanation for the differing results is that this change is not correctly detected if the code is not recompiled.

RQ3: How do C# Compiler settings influence the energy consumption of software systems?
By analysing the compiler settings offered by .NET Core, we found one setting that should be avoided as it significantly increases the energy consumption and execution time. This is the combination of QuickJIT, QuickJIT for loops and Tiered compilation. We also verified that it is not possible to combine QuickJIT with the ReadyToRelease setting. Compilations using these settings create two distinct sets of codes, with the behaviour of each set equal to one of the settings. Finally, for all other settings, we found that most settings have significant differences in energy consumption, however, these differences were not consistent across the different benchmarks. Thus, we cannot point to one setting as the optimal setting.

By looking at the energy consumption per second, we found that all settings showed a similar energy consumption. This points to a correlation between energy consumption and execution time, with longer execution times showing higher energy consumption.

RQ4: How consistent is the impact of compiler settings across hardware architectures?
By performing the measurements for the compiler settings on multiple laptops, we found that the overall trends were consistent. The exact energy consumption differed between the devices, but the relationships between the settings were the same.

RQ5: How do functionally equal programming choices impact energy consumption?
We provide a number of code examples that can be used to perform experiments. As we did not perform such experiments, we cannot provide an answer to this research question.

RQ: How can the energy consumption of software systems be reduced?
Overall, we found that there are a lot of factors that can influence the energy consumption of software systems. Ensuring that a system is using power-save settings can reduce energy consumption. Furthermore, the choice of compilation setting can have serious impacts on energy consumption. Although we did not manage to identify the best choice, we did find that the QLT setting should be avoided. It is advisable to perform measurements to find the best choice for a software system and to ensure that changing a hardware setting causes the intended result.

10.3 Recommendations

There are several avenues that require further study. First of all, the code in the benchmarks used in this study can be studied to discover if there are code segments that can be linked to the differences in energy consumption observed with the compiler settings. If such links exist and can be found, guidelines can be created to indicate when certain compiler settings should or should not be used.

Furthermore, the analysis of the compiler settings could be repeated with a different, larger codebase. Since the benchmarks used in our experiments were created to compare the performance of programming languages, they are of limited size.

Another possibility is to repeat the experiments on a different operating system. This could be Windows or perhaps even a mobile environment. Performing tests on Windows would require a different measurement method or the development of a RAPL driver for Windows. A mobile environment would also require a different measurement method.

It is also possible to perform experiments based on the programming choices suggested in Chapter 9. Research on the differences in energy consumption of different programming choices is required before best practice guidelines can be created. The creation of best practice guidelines will enable developers to adapt their programming to reduce the energy consumption of their software.

Finally, since the benchmarks we used to perform our experiments are also available for different programming languages, our research could be repeated with different programming languages to determine if the hardware settings show a similar impact. Where available, our methodology can be used to compare different compiler options of other programming languages.

Bibliography

[1] L. M. Hilty, V. Coroama, M. O. de Eicker, T. Ruddy, and E. Müller, "The role of ict in energy consumption and energy efficiency," *Report to the European Commission, DG INFSO, Project ICT ENSURE: European ICT Sustainability Research, Graz University*, vol. 1, pp. 1–60, 2009.

[2] J. Malmodin, A. s. Moberg, D. Lundén, G. Finnveden, and N. Lövehagen, "Greenhouse gas emissions and operational electricity use in the ict and entertainment & media sectors," *Journal of Industrial Ecology*, vol. 14, no. 5, pp. 770–790, 2010. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1530-9290.2010.00278.x

[3] A. S. G. Andrae, "Projecting the chiaroscuro of the electricity use of communication and computing from 2018 to 2030," *Researchgate. net*, 2019.

[4] M. C. Sanchez, R. E. Brown, C. Webber, and G. K. Homan, "Savings estimates for the united states environmental protection agency's energy star voluntary product labeling program," *Energy Policy*, vol. 36, no. 6, pp. 2098 – 2108, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0301421508001092

[5] B. Urban, K. Roth, M. Singh, and D. Howes, "Energy consumption of consumer electronics in u.s. homes in 2017," 12 2017.

[6] M. Avgerinou, P. Bertoldi, and L. Castellazzi, "Trends in data centre energy consumption under the european code of conduct for data centre energy efficiency," *Energies*, vol. 10, no. 10, 2017. [Online]. Available: https://www.mdpi.com/1996-1073/10/10/1470

[7] T. A. Ghaleb, "Software energy measurement at different levels of granularity," in *2019 International Conference on Computer and Information Sciences (ICCIS)*, 04 2019, pp. 1–6.

[8] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, 08 2010, pp. 189–194.

[9] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, “Rapl in action: Experiences in using rapl for power measurements,” *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, Mar. 2018. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/3177754

[10] L. Cruz and R. Abreu, “Catalog of energy patterns for mobile applications,” *Empirical Software Engineering*, 03 2019.

[11] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. a. Saraiva, “Towards a green ranking for programming languages,” in *Proceedings of the 21st Brazilian Symposium on Programming Languages*, ser. SBLP 2017. New York, NY, USA: ACM, 2017, pp. 7:1–7:8. [Online]. Available: http://doi.acm.org/10.1145/3125374.3125382

[12] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, “Energy efficiency across programming languages: How do energy, time, and memory relate?” in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: ACM, 2017, pp. 256–267. [Online]. Available: http://doi.acm.org/10.1145/3136014.3136031

[13] R. Pereira, M. Couto, J. a. Saraiva, J. Cunha, and J. a. P. Fernandes, “The influence of the java collection framework on overall energy consumption,” in *Proceedings of the 5th International Workshop on Green and Sustainable Software*, ser. GREENS’16. New York, NY, USA: ACM, 2016, pp. 15–21. [Online]. Available: http://doi.acm.org/10.1145/2896967.2896968

[14] G. Melfe, A. Fonseca, and J. a. P. Fernandes, “Helping developers write energy efficient haskell through a data-structure evaluation,” in *Proceedings of the 6th International Workshop on Green and Sustainable Software*, ser. GREENS ’18. New York, NY, USA: ACM, 2018, pp. 9–15. [Online]. Available: http://doi.acm.org/10.1145/3194078.3194080

[15] C. Bunse and S. Stiemer, “On the energy consumption of design patterns,” *Softwaretechnik-Trends*, vol. 33, pp. 7–8, 05 2013.

[16] E. Jagroep, J. M. E. M. van der Werf, S. Jansen, M. Ferreira, and J. Visser, “Profiling energy profilers,” in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC ’15. New York, NY, USA: ACM, 2015, pp. 2198–2203. [Online]. Available: http://doi.acm.org/10.1145/2695664.2695825

[17] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, “Measuring energy consumption for short code paths using rapl,” *SIGMETRICS Perform.*

Eval. Rev., vol. 40, no. 3, p. 13–17, Jan. 2012. [Online]. Available: https://doi.org/10.1145/2425248.2425252

[18] S. Desrochers, C. Paradis, and V. M. Weaver, "A validation of dram rapl power measurements," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 455–470. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/2989081.2989088

[19] K. Liu, G. Pinto, and Y. D. Liu, "Data-oriented characterization of application-level energy optimization," in *Fundamental Approaches to Software Engineering*, A. Egyed and I. Schaefer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 316–331.

[20] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva, "Helping programmers improve the energy efficiency of source code," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 05 2017, pp. 238–240.

[21] D. Beyer and P. Wendler, "Cpu energy meter: A tool for energy-aware algorithms engineering," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds. Cham: Springer International Publishing, 2020, pp. 126–133.

[22] Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, J. Penhoat, and L. Seinturier, "Taming energy consumption variations in systems benchmarking," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 36–47. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/3358960.3379142

[23] R. Pereira, P. Simão, J. Cunha, and J. a. Saraiva, "jstanley: Placing a green thumb on java collections," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 856–859. [Online]. Available: http://doi.acm.org/10.1145/3238147.3240473

[24] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 225–236. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884869

[25] G. Pinto, K. Liu, F. Castor, and Y. D. Liu, “A comprehensive study on the energy efficiency of java’s thread-safe collections,” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 10 2016, pp. 20–31.

[26] A. R. Tonini, L. M. Fischer, J. C. B. d. Mattos, and L. B. d. Brisolara, “Analysis and evaluation of the android best practices impact on the efficiency of mobile applications,” in *2013 III Brazilian Symposium on Computing Systems Engineering*, 12 2013, pp. 157–158.

[27] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes, “Haskell in green land: Analyzing the energy behavior of a purely functional language,” in *SANER*. IEEE Computer Society, 2016, pp. 517–528.

[28] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes, “On haskell and energy efficiency,” *Journal of Systems and Software*, vol. 149, pp. 554 – 580, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218302747

[29] G. Melfe, A. Fonseca, and J. a. P. Fernandes, “Evaluation of the impact on energy consumption of lazy versus strict evaluation of haskell data-structures,” in *Proceedings of the XXII Brazilian Symposium on Programming Languages*, ser. SBLP ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 83–89. [Online]. Available: https://doi.org/10.1145/3264637.3264648

[30] K. Chantarasathaporn and C. Srisa-an, “Object-oriented programming strategies in c# for power conscious system,” *International Journal of Computer and Information Engineering*, vol. 1, no. 10, pp. 3198 – 3203, 2007. [Online]. Available: https://publications.waset.org/vol/10

[31] A. Litke, K. Zotos, A. Chatzigeorgiou, and G. Stephanides, “Energy consumption analysis of design patterns,” in *Proceedings of the International Conference on Machine Learning and Software Engineering*, 2005, pp. 86–90.

[32] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, “Initial explorations on design pattern energy usage,” in *2012 First International Workshop on Green and Sustainable Software (GREENS)*, 06 2012, pp. 55–61.

[33] D. Feitosa, R. Alders, A. Ampatzoglou, P. Avgeriou, and E. Y. Nakagawa, “Investigating the effect of design patterns on energy consumption,” *Journal of Software: Evolution and Process*, vol. 29, no. 2, p. e1851, 2017, e1851 JSME-16-0030.R2. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1851

[34] A. Noureddine and A. Rajan, “Optimising energy consumption of design patterns,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE ’15. IEEE Press, 2015, p. 623–626.

[35] G. Agosta, M. Bessi, E. Capra, and C. Francalanci, “Dynamic memoization for energy efficiency in financial applications,” in *2011 International Green Computing Conference and Workshops*, 07 2011, pp. 1–8.

[36] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, “Measuring energy and power with papi,” in *2012 41st International Conference on Parallel Processing Workshops*, 2012, pp. 262–268.

Appendix A

CSV example

A visualisation of a CSV file produced when running the benchmark tests can be found in Table A.1. The values have been rounded to two decimals to make it legible and to make it fit on the page. Values reported by RAPL are stored with fourteen decimal values. The CSV file does not include the headers indicating for the columns and uses semicolons to separate values. The values used are the results of a test executed in power-save without any C# compiler options.

Table A.1: A visualisation of a CSV file containing the results of running the set of benchmarks. Values rounded to two decimals for legibility.

Name	PKG J	PP0 J	PP1 J	DRAM J	Psys J	Start Temp	End Temp	Time ms
binary-trees	279.87	250.81	0.00	11.05	375.74	36.00	54.00	9 701.77
binary-trees	264.21	235.01	0.00	11.11	362.83	54.00	52.00	9 928.49
binary-trees	266.59	238.15	0.00	10.73	362.74	52.00	53.00	9 621.90
binary-trees	282.79	252.84	0.00	11.33	385.43	53.00	55.00	10 007.40
binary-trees	270.88	243.15	0.00	10.40	366.52	55.00	80.00	9 328.66
binary-trees	275.88	245.99	0.00	11.32	379.58	80.00	56.00	10 034.40
binary-trees	272.35	244.17	0.00	10.68	373.32	56.00	55.00	9 474.38
binary-trees	269.01	241.59	0.00	10.27	367.68	55.00	54.00	9 249.91
binary-trees	254.06	224.54	0.00	11.03	355.61	54.00	57.00	9 997.99
binary-trees	278.98	249.78	0.00	11.25	380.83	57.00	57.00	9 782.01
fasta	59.08	56.46	0.00	0.68	72.26	36.00	73.00	898.99
fasta	56.74	54.13	0.00	0.69	70.23	73.00	72.00	909.06
fasta	53.06	50.45	0.00	0.69	66.42	72.00	70.00	921.12
fasta	52.99	50.11	0.00	0.75	67.57	70.00	71.00	1 001.88
fasta	52.59	49.86	0.00	0.73	66.78	71.00	74.00	970.62
fasta	53.11	50.35	0.00	0.72	67.71	74.00	72.00	968.78
fasta	50.24	47.52	0.00	0.72	64.30	72.00	69.00	951.16

fasta	52.72	49.87	0.00	0.73	67.35	69.00	76.00	981.43
fasta	50.74	47.82	0.00	0.74	65.43	73.00	67.00	985.15
fasta	51.18	48.16	0.00	0.78	66.58	67.00	71.00	1 039.94
fannkuch-redux	341.30	325.58	0.00	3.77	426.03	41.00	58.00	5 595.84
fannkuch-redux	362.80	343.40	0.00	4.52	469.32	58.00	65.00	6 917.38
fannkuch-redux	345.48	326.13	0.00	4.54	456.04	65.00	64.00	6 949.73
fannkuch-redux	290.99	274.30	0.00	3.93	386.05	64.00	66.00	6 003.50
fannkuch-redux	336.71	317.50	0.00	4.50	448.71	66.00	63.00	6 884.07
fannkuch-redux	336.10	317.00	0.00	4.48	447.80	63.00	66.00	6 848.29
fannkuch-redux	285.87	268.83	0.00	4.00	382.06	66.00	62.00	6 123.06
fannkuch-redux	286.10	269.14	0.00	3.98	381.74	62.00	65.00	6 088.90
fannkuch-redux	284.94	267.97	0.00	3.99	380.60	65.00	67.00	6 102.58
fannkuch-redux	283.25	266.21	0.00	3.98	378.49	66.00	63.00	6 093.54
k-nucleotide	181.22	168.73	0.00	4.53	234.44	39.00	53.00	4 161.15
k-nucleotide	171.81	159.91	0.00	4.60	224.44	53.00	60.00	3 929.41
k-nucleotide	173.46	160.66	0.00	4.68	229.33	60.00	60.00	4 259.53
k-nucleotide	167.08	154.49	0.00	4.57	224.65	60.00	65.00	4 269.76
k-nucleotide	157.61	145.69	0.00	4.41	214.76	65.00	61.00	4 038.43
k-nucleotide	175.33	162.15	0.00	4.67	237.56	61.00	60.00	4 531.39
k-nucleotide	158.25	146.81	0.00	4.38	214.16	60.00	57.00	3 861.12
k-nucleotide	158.53	147.07	0.00	4.33	216.13	57.00	59.00	3 848.98
k-nucleotide	159.53	147.84	0.00	4.39	217.28	59.00	68.00	3 980.61
k-nucleotide	162.16	148.80	0.00	4.63	224.79	68.00	56.00	4 582.18
n-body	97.80	87.51	0.00	2.44	126.52	38.00	71.00	3 720.57
n-body	101.45	91.01	0.00	2.47	131.18	71.00	53.00	3 766.13
n-body	100.74	90.26	0.00	2.49	131.16	53.00	53.00	3 799.66
n-body	98.57	88.02	0.00	2.51	132.35	54.00	52.00	3 838.95
n-body	100.43	90.14	0.00	2.47	136.38	52.00	77.00	3 782.99
n-body	97.07	86.45	0.00	2.53	134.49	77.00	52.00	3 873.14
n-body	99.01	88.71	0.00	2.49	135.34	52.00	52.00	3 795.31
n-body	98.96	88.60	0.00	2.47	135.99	52.00	76.00	3 773.11
n-body	98.81	88.54	0.00	2.47	134.83	76.00	52.00	3 776.92
n-body	98.45	87.78	0.00	2.53	135.84	52.00	51.00	3 861.67
pidigits	24.23	21.96	0.00	0.54	30.92	36.00	45.00	825.31
pidigits	23.97	21.78	0.00	0.53	30.43	45.00	49.00	806.06
pidigits	22.97	20.74	0.00	0.53	29.09	49.00	49.00	796.59
pidigits	23.58	21.27	0.00	0.56	30.12	49.00	49.00	846.74
pidigits	22.54	20.27	0.00	0.54	28.83	49.00	51.00	819.88
pidigits	22.74	20.46	0.00	0.54	29.19	51.00	52.00	813.15

pidigits	22.19	19.84	0.00	0.55	29.40	52.00	50.00	825.74
pidigits	24.25	22.07	0.00	0.53	30.80	50.00	48.00	799.66
pidigits	22.88	20.58	0.00	0.55	29.67	48.00	51.00	831.20
pidigits	23.73	21.52	0.00	0.53	30.26	51.00	74.00	800.14
reverse-complement	23.00	20.73	0.00	0.92	30.80	36.00	50.00	685.29
reverse-complement	21.90	19.76	0.00	0.90	28.71	50.00	49.00	709.37
reverse-complement	22.15	20.05	0.00	0.89	28.79	49.00	52.00	694.72
reverse-complement	21.68	19.44	0.00	0.93	28.89	52.00	53.00	756.22
reverse-complement	22.61	20.59	0.00	0.86	29.21	52.00	55.00	657.24
reverse-complement	21.89	19.68	0.00	0.91	28.85	55.00	50.00	713.41
reverse-complement	22.14	20.09	0.00	0.87	28.77	50.00	52.00	671.51
reverse-complement	21.99	19.84	0.00	0.89	28.79	52.00	52.00	696.39
reverse-complement	21.99	19.89	0.00	0.88	28.75	52.00	52.00	675.41
reverse-complement	21.13	19.05	0.00	0.88	27.72	52.00	72.00	676.20
spectral-norm	20.90	19.95	0.00	0.24	25.51	39.00	56.00	348.96
spectral-norm	17.64	16.54	0.00	0.27	22.37	56.00	56.00	405.46
spectral-norm	19.04	18.02	0.00	0.25	23.77	56.00	59.00	369.42
spectral-norm	19.99	19.07	0.00	0.23	24.74	59.00	60.00	336.54
spectral-norm	19.90	18.93	0.00	0.23	24.76	60.00	60.00	339.64
spectral-norm	19.50	18.58	0.00	0.22	24.28	60.00	61.00	334.82
spectral-norm	19.59	18.65	0.00	0.23	24.56	61.00	61.00	343.60
spectral-norm	17.07	15.88	0.00	0.29	22.49	61.00	60.00	440.57
spectral-norm	19.20	18.25	0.00	0.23	24.25	60.00	61.00	345.49
spectral-norm	17.13	15.94	0.00	0.29	22.43	61.00	73.00	429.77
regex-redux	472.48	442.20	0.00	8.13	605.37	36.00	51.00	10 860.10
regex-redux	428.73	398.56	0.00	8.09	572.23	51.00	58.00	10 804.90
regex-redux	409.00	379.33	0.00	7.93	553.29	58.00	54.00	10 564.20
regex-redux	426.57	396.65	0.00	7.99	575.61	54.00	58.00	10 640.80
regex-redux	439.50	407.63	0.00	8.47	594.27	58.00	52.00	11 387.80
regex-redux	415.50	385.02	0.00	8.13	563.10	52.00	79.00	10 867.70
regex-redux	394.67	366.05	0.00	7.70	536.76	79.00	51.00	10 190.50
regex-redux	398.48	369.88	0.00	7.70	541.72	51.00	54.00	10 181.70
regex-redux	408.35	376.40	0.00	8.50	559.62	54.00	57.00	11 397.10
regex-redux	431.15	397.75	0.00	8.82	589.38	57.00	57.00	11 915.70
mandelbrot	110.64	105.84	0.00	1.15	137.63	38.00	62.00	1 710.04
mandelbrot	103.30	98.35	0.00	1.18	131.03	62.00	70.00	1 762.82
mandelbrot	98.59	93.41	0.00	1.24	126.87	70.00	61.00	1 846.45
mandelbrot	96.91	91.79	0.00	1.21	125.23	61.00	63.00	1 808.23
mandelbrot	95.50	90.23	0.00	1.23	123.75	63.00	71.00	1 836.15

mandelbrot	93.30	87.95	0.00	1.26	122.10	71.00	74.00	1 874.35
mandelbrot	91.71	86.50	0.00	1.25	120.71	72.00	66.00	1 864.42
mandelbrot	91.48	86.29	0.00	1.25	120.58	66.00	74.00	1 871.93
mandelbrot	91.22	86.03	0.00	1.25	120.96	74.00	73.00	1 870.72
mandelbrot	89.73	84.31	0.00	1.30	120.98	73.00	66.00	1 946.47

Appendix B

Data Analysis Chart Examples

An example of the bar chart with error bars to indicate confidence intervals can be found in Figure B.1. The blue bar represents the PKG domain with all executions. The orange bar represents the PKG domain, using only the last 4 executions of every run. The grey bar represents the Psys domain with all executions. The yellow bar represents the Psys domain, using only the last 4 executions of every run.

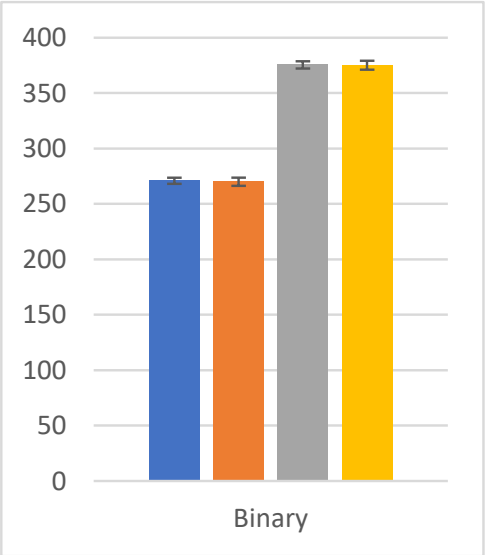

Figure B.1: Results for the binary benchmark with Tiered Compilation

An example of the chart depicting the energy consumption per time unit can be found in Figure B.2 The blue bar represents the Psys domain divided by the execution time with all executions. The red bar represents the Psys domain divided by the execution time, using only the last 4 executions of every run.

An example of a chart with combined averages and combined error bars can be found in Figure B.3.

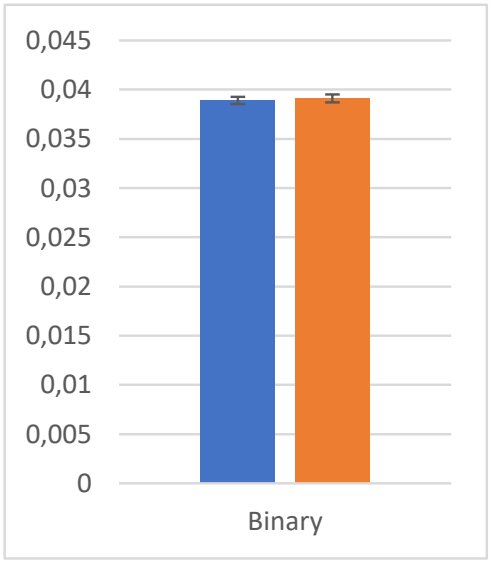

Figure B.2: The energy consumption per millisecond for the binary benchmark with Tiered Compilation

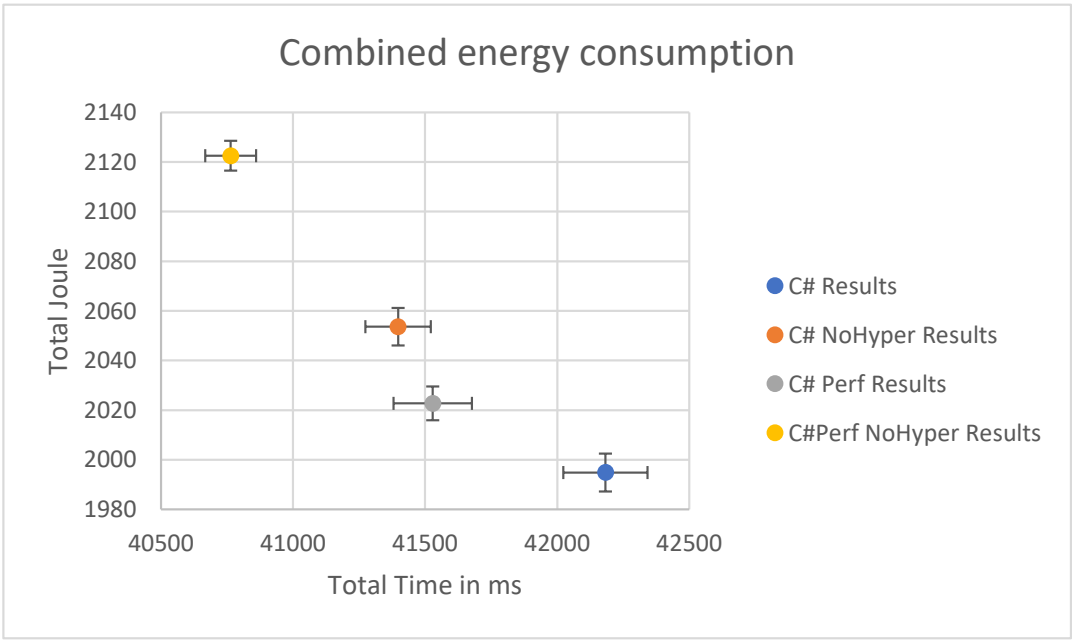

Figure B.3: Combined chart for the initial hardware measurements

Appendix C

Toll Calculations

The complete code demonstrating the power of pattern matching from the presentation by Bill Wagner at NDC London 2020[1] [2].

```
using System;

namespace NDCLondon
{
    public static class tollCalculations
    {

        private static bool IsWeekDay(DateTime timeOfToll) =>
            timeOfToll.DayOfWeek switch
            {
                DayOfWeek.Saturday => false,
                DayOfWeek.Sunday => false,
                _ => true
            };

        private enum TimeBand
        {
            MorningRush,
            Daytime,
            EveningRush,
            Overnight
        }

        private static TimeBand GetTimeBand(DateTime timeOfToll)
```

[1] Change your habits: Modern techniques for modern C#
[2] Change your habits: Modern techniques for modern C# - Bill Wagner

```
{
    int hour = timeOfToll.Hour;
    if (hour < 6)
        return TimeBand.Overnight;
    else if (hour < 10)
        return TimeBand.MorningRush;
    else if (hour < 16)
        return TimeBand.Daytime;
    else if (hour < 20)
        return TimeBand.EveningRush;
    else
        return TimeBand.Overnight;
}



public static decimal PeakTimePremiumImperative
        (DateTime timeOfToll, boolean inbound)
{
    if (IsWeekDay(timeOfToll))
    {
        if(inbound)
        {
            var timeBand = GetTimeBand(timeOfToll);
            if (timeBand == TimeBand.MorningRush)
            {
                return 2.00m;
            }
            else if (timeBand == TimeBand.Daytime)
            {
                return 1.50m;
            }
            else if (timeBand == TimeBand.EveningRush)
            {
                return 1.00m;
            }
            else
            {
                return 0.75m;
```

```
            }
        }
        else
        {
            var timeBand = GetTimeBand(timeOfToll);
            if (timeBand == TimeBand.MorningRush)
            {
                return 1.00m;
            }
            else if (timeBand == TimeBand.Daytime)
            {
                return 1.50m;
            }
            else if (timeBand == TimeBand.EveningRush)
            {
                return 2.00m;
            }
            else
            {
                return 0.75m;
            }
        }
    }
    else
    {
        return 1.00m
    }
}

public static decimal PeakTimePremiumPattern
            (DateTime timeOfToll, boolean inbound) =>
(IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush,    true)   => 2.00m,
        (true, TimeBand.MorningRush,    false)  => 1.00m,
        (true, TimeBand.Daytime,        true)   => 1.50m,
        (true, TimeBand.Daytime,        false)  => 1.50m,
        (true, TimeBand.EveningRush,    true)   => 1.00m,
        (true, TimeBand.EveningRush,    false)  => 2.00m,
```

```
                (true, TimeBand.Overnight,      true)   => 0.75m,
                (true, TimeBand.Overnight,      false)  => 0.75m,
                (false, TimeBand.MorningRush,   true)   => 1.00m,
                (false, TimeBand.MorningRush,   false)  => 1.00m,
                (false, TimeBand.Daytime,       true)   => 1.00m,
                (false, TimeBand.Daytime,       false)  => 1.00m,
                (false, TimeBand.EveningRush,   true)   => 1.00m,
                (false, TimeBand.EveningRush,   false)  => 1.00m,
                (false, TimeBand.Overnight,     true)   => 1.00m,
                (false, TimeBand.Overnight,     false)  => 1.00m,
            };

        public static decimal PeakTimePremiumPatternReduced
                    (DateTime timeOfToll, boolean inbound) =>
        (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
            {
                (true, TimeBand.MorningRush,    true)   => 2.00m,
                (true, TimeBand.MorningRush,    false)  => 1.00m,
                (true, TimeBand.Daytime,        _)      => 1.50m,
                (true, TimeBand.EveningRush,    true)   => 1.00m,
                (true, TimeBand.EveningRush,    false)  => 2.00m,
                (true, TimeBand.Overnight,      _)      => 0.75m,
                (false, _,                      _)      => 1.00m,
            };
    }
}
```

Listing 6: The complete code with conditional statements and two pattern matching functions