Summary

This project is a Master thesis written by two students at Aalborg University. The subject of the project is programming technology and more specifically, energy aware programming.

The aim of this project is to help programmers be more aware of the energy usage of their code. The motivation for this is the company Edora which wants to monitor their energy consumption. This project focuses on C# since this area is lacking when it comes to energy consumption research.

The goal of the project is to create a Continuous Integration pipeline which utilizes an energy testing framework for running tests and reporting energy improvements or degradations to the developer.

To achieve this goal, research is made into the related fields, such as the Intel RAPL functionality and software testing in general. The current state of Continuous Integration and overnight builds are discussed in regards to where the energy testing should be performed. Additionally, the relevant mathematical topics required to understand and analyse the results of energy testing is researched. These topics include different types of distributions as well as their skewness along with statistical significance tests and potential problems with these.

This research is used to facilitate the creation of an energy measuring framework which is built into MSTestV2 to allow custom tests to be run multiple times in order to create energy consumption distributions.

This framework is first tested on a simple Fibonacci function to compare the energy readings of production code and the testing code. Here discrepancies are discovered which starts an investigation into the limitations of Intel RAPL and how small tests can be measured and still achieve accurate results.

The Intel RAPL readings are adapted to shorter benchmarks by measuring multiple code runs at one time and averaging the run time and energy consumption over multiple runs, to prevent inaccurate readings and 0-readings.

With the knowledge of the Intel RAPL limitations, a test is performed on a larger system, namely the Newtonsoft.JSON library where the energy testing framework is proven to be able to identify time and energy optimisations. In an attempt to replicate these results with a different test, complications are discovered which means that the second tested change is only available on Windows and can therefor not be tested since the testing framework requires a Linux system in order to utilize the Intel RAPL readings. However, a discrepancy in the codebase of Newtonsoft.JSON is discovered through this testing, which means that the newest version of Newtonsoft.JSON is using legacy code which through an additional experiment is proven to be at a cost of both time and energy compared to native .NET implementations.

The initial goal of integrating this energy testing framework into a Continuous

Integration toolchain is not reached in this project, but the ground work is laid out. This ground work would allow for the creation of an overnight energy testing framework to be built. However, a Continuous Integration tool would require additional research and experimentation into minimizing the run times and iterations of energy tests.

Energy measurements of tests

Exploring the energy consumption of tests in C# using Intel RAPL

Jonas Bylling Andersen, Phillip Bengtson Jørgensen

Software, cs-22-pt-10-03, 2022-06

Master's Project

Report written in LaTeX. Figures are mainly created using Matplotlib

Department of Computer Science
Aalborg University
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Energy measurements of tests

Theme:
Programming Technology

Project Period:
Spring Semester 2022

Project Group:
cs-22-pt-10-03

Participant(s):
Jonas Bylling Andersen
Phillip Bengtson Jørgensen

Supervisor(s):
Bent Thomsen
Thomas Bøgholm

Copies: 1

Page Numbers: 82

Date of Completion:
June 10th, 2022

Abstract:

Currently there is a focus on energy awareness. This report focuses on performing energy measurements on tests, with the goal of integrating these measurements into a Continuous Integration. We do this by integrating energy measurements into the MSTest framework. Upon investigation we find that measurements are often inaccurate and therefore test the limitations of Intel RAPL with regards to sampling rate. From this knowledge we establish a minimum testing duration. We detect energy differences in Newtonsoft.JSON-framework through unit tests. The first being for Integer serialization and the second being for path traversal. We are able to measure the energy consumption of tests and are able to detect differences in energy consumption and run time between the changes introduced to the code. The contribution is methodology regarding how the changes in energy usage of tests can be measured as well as how smaller tests can be accurately measured using Intel RAPL.

Contents

Preface

This project is a Master thesis on the 4th semester of the Master of Software at Aalborg University. This project spans from 1st of February 2022 to 10th of June 2022. The project is supervised by Bent Thomsen[1] and Thomas Bøgholm[2].

Aalborg University, June 10th, 2022

Jonas Bylling Andersen
<jban18@student.aau.dk>

Phillip Bengtson Jørgensen
<pjarge17@student.aau.dk>

[1] https://vbn.aau.dk/da/persons/110568
[2] https://vbn.aau.dk/da/persons/112525

Chapter 1

Introduction

1.1 Motivation

Developers and companies have an interest in increasing awareness of their energy consumption. Edora[1] is an example of such a company. From previous contact, they have voiced their concern for their own energy consumption. Furthermore, they already monitor their energy consumption in their production environment. The current way which Edora monitors their energy consumption is by recording the energy consumption of their entire server setup and then creating a metric of the energy consumed by dividing energy usage by transactions, to find the energy usage per transaction. In this project the goal is to create a solution for improving insight regarding the energy consumption of a code base. Edora uses C# for their code, but a large number of articles written within the energy awareness field focus on Java, which makes it hard for Edora to apply these methods to their C# code. As there is a lack of projects focused on C#, it will be the focus of this project.

1.2 Goal of the project

The goal of this project is to create an energy testing framework which can be implemented in a Continuous Integration pipeline. This is with the purpose of easing the access to energy measurements for programmers and provide them with insight into the energy impact of their code changes.

[1] `https://edora.dk/`

Chapter 2

Related works

In this chapter we establish the current knowledge relevant to the project. This knowledge is mainly focused on energy awareness tools, benchmarking, energy metrics, common developer practices and data processing.

2.1 Energy Benchmarking With Doom

This report builds on the knowledge from the previous work in the report *"Energy Benchmarking With Doom"*[16]. In [16], Nielsen et al. tested how programming languages compare in terms of energy consumption on the video game Doom. Before being able to conduct any benchmarks, there is a major section on how RAPL works and it's accuracy when compared to external energy meters. In order to perform benchmarking with Doom a tool called rapl.rs was created. This tool logs the RAPL-values for a generic program by executing the program and utilizing a separate logging thread. Upon researching the results of benchmarking the different Doom implementations, it was discovered that the winners of consuming the least amount of energy were C and C++, with C# close after. It was also discovered that the Java implementation had a major architectural difference which amounted to Java having the worst performance in terms of energy consumption.

2.2 IDE Extension for Reasoning About Energy Consumption

In [17], Nørhave et al. work on creating an extension for the source-code editor Visual Studio Code. The extension is able to measure energy consumption of code snippets using two different approaches. The first method being an estimate given based on the number of instructions of the measured snippet. In order to evaluate the codesnippet, the authors create an interpreter of CIL code to count the specific

number of instructions which they can then create an energy estimate from. This estimate is rather fast, but has inaccuracies. In order to create accurate estimates, the second approach is used. In this approach the codesnippet is run and measured using RAPL. This method takes quite a lot longer since it may come with noise. In order to combat the noise, the authors evaluates the method using Cochran's formula to estimate the number of executions needed to remove the noise. The authors also evaluate the accuracy of their estimation engine using machine learning models. In their analysis, the best model is the random forest machine learning model which has a minimum of -7.49% and a maximum of 9.19% estimation error. The median of this model is 1.06% - so a slight overestimation.

The main take away from this paper is their two approaches for doing energy estimations. Given the target user of the system is developers, the static estimation method is favorable since doing the dynamic estimation requires a lot more time.

2.3 Robust benchmarking in noisy environments

In [2], Jiahao Chen and Jarrett Revels present a strategy for benchmarking in noisy environments. Additionally, the authors implement their solution for BechmarkTools[1] for Julia, in the JuliaCI's continuous integration pipeline. The authors presents a large amount of papers regarding how many variations of a running system can affect a benchmark. For example, garbage collection, CPU frequency scaling, memory layout and compiler optimizations. The authors have created an algorithm that can estimate the number of executions in order to defeat timer error.

The authors of this paper do in large part what this report also aims to accomplish; executing continuous integration pipelines while recording metrics of the processes. The difference between the paper and the goal of this project is that instead of recording these metrics of time, the goal is to record the energy consumption.

2.4 The hunt for the guzzler: Architecture-based energy profiling using stubs

In [10] E. Jagroep et al. use a method called stubbing for identifying the energy use of individual parts of a program. Stubbing is the act of replacing part of the code with a simple method which returns a static value similar to what would normally be returned, without any computations involved on the stub. For energy measuring E. Jagroep et al measure the energy use of a program, before and after substituting part of the code with a stub. This is done in order to determine the energy use of the stubbed code, by comparing the energy of the run with the

[1] `https://github.com/JuliaCI/BenchmarkTools.jl`

section included to the run without the code. By doing stubbing, the authors gain knowledge regarding if the stubbed part of the program is the "energy guzzler" of the system.

The use of stubbing in code is not a new approach, but E. Jagroep et al use this method for testing energy use of individual parts, where it previously is mostly used for missing features and for test cases. They describe their objective with the paper as providing a way for software architects to gain information about a qualitative aspect of their code, which is the energy consumption.

The results they find show that it is possible to use stubs to estimate energy usage of individual parts of code, and even identify energy hot spots in the code. Additionally, they recommend the creation of automatically generated stubs as a part of future work in the area.

2.5 Coppers - Rust energy measurements

Students from Delft technical university have made a framework for measuring changes in energy usage from one version of a program to another[20]. This test framework is written in Rust and uses the Rust nightly toolchain, as it relies on unstable features in the Rust compiler[11][2].

When using Coppers, it will run the projects unit tests and give feedback on the energy usage of each test, which can then be compared to the energy results of the previous release of the given project. It is also possible to mark tests as ignore, in which case the test will not be executed and measured. Additionally, the energy usage results can be plotted visually, by enabling their visualisation feature, which is disabled by default.

The tests are only run once each, and the developers also describe how the results can be imprecise and they provide measures to help users of their framework get as accurate a reading as possible. The measures they describe include closing background processes and disabling potential interrupts through notifications, as well as advising the use of the same hardware system when comparing different versions.

2.6 PLATYPUS: Software-based Power Side-Channel Attacks on x86

In this article[13], Moritz Lipp et al. explore the significance of being able to access power consumption of a system using Intel RAPL. While the RAPL documentation claims a 1ms update interval, the authors dicover that they are able to sample

[2]The Rust nightly toolchain is a daily build of the master code, which happens overnight, hence the name "nightly".

substantially faster at up to 50 μs or 150 μs. The authors discover that measuring directly on each x86-instruction, they are able to distinguish different operations from one another. Further more, they are able to distinguish values of the variables in the x86-instructions. In the article the authors are able to recover a RSA key pair with a 512 bit modulus in 211 minutes.

The main take-away from this article is that by sampling the same functionality over and over again, the authors are able to very precisely measure the energy consumption of the system, enough to even distinguish the values of the variables to the CPU.

2.7 The impact of Platypus attack on RAPL

In section 2.6, the research paper for Platypus was discussed. However, this had an impact on functionality of Intel RAPL. In [9], Intel describes how they handled the attack. Firstly, in the Linux system, the default permissions to gain information on energy consumption has been changed. Additionally, Intel has released two microcode patches which modifies the energy information reported by RAPL. Specifically, Intel has implemented a filter for the data which is available to the user. This filters status is dependant on the status of Intel SGX. The first microcode patch adds the filter, which approximates the energy consumption instead of reading directly from the power circuit of the CPU. The second microcode patch changes RAPL's update frequency and adds some random energy noise. In the example in [9], the unfiltered version consumes 169J in a program with data A and 171J in the same program with data B. The filtered version of the same program gives a range between 165J and 185J, regardless of which data is being used. This filtering of the output of Intel RAPL can however be turned on and off. Either via disabling security features in the BIOS of the computer or by writing to a model specific register on the CPU. In Figure 2.1, the solution Intel implemented in the microcode can be seen.

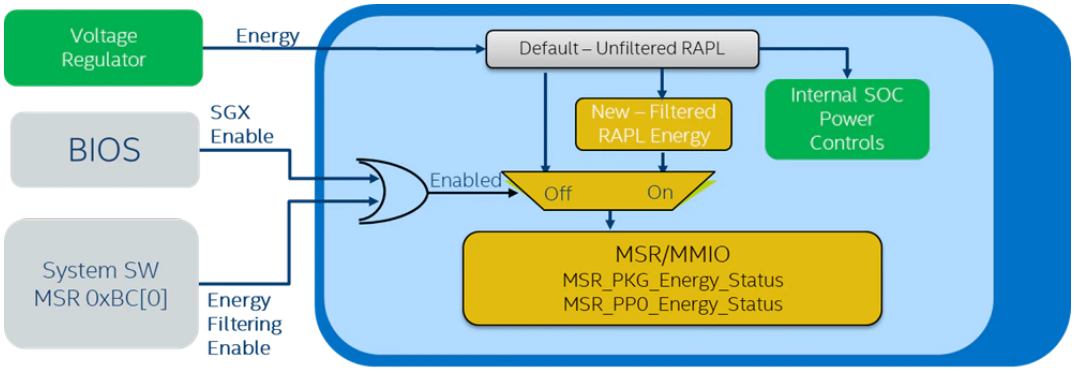

Figure 2.1: Overview of Intel RAPL filtering, Source [9]

2.8 Software Engineering

In [19], Sommerville covers all aspects of software engineering. The relevant part of this book, for this project, is Chapter 8 about Software testing. In this chapter, Sommerville includes a definition of development testing, which is the testing carried out by the team developing the system. Sommerville further breaks this into three stages of testing: Unit testing, Component testing and System testing. The Unit testing is tests of individual program units or objects classes. Component testing is a collection of several units integrated to create composite components. Finally, system testing is a collection of some or all components of a system and the system is tested as a whole.

The important knowledge from this source is, as testing of energy consumption is performed, we need to know how large a component of the system we are testing. The scope of testing could potentially have a large impact on the energy consumption of the tests conducted.

2.9 Can We Spot Energy Regressions using Developers Tests?

In [4], Dangot et al. examine the possibility of utilizing the developers continuous integration in order to perform energy regression testing.

The authors suggests specific tools for Java for conducting the testing and measuring the energy impact of the unit tests. The study of the paper will according to the authors pave the way for automated regression testing of software energy consumption. Unfortunately the paper does not conduct experiments into how well the method is compared to the actual consumption of the program under test. In their threats to validity the authors do however mention that unit test may not be a strong representation of the production workload. In order to handle this issue, the authors mention that the energy consumption they measure is weighed down by the number of lines executed. Overall the method presented seems like a good indication of how to conduct such automated energy regression testing, but the method needs to be tested in order to prove it's validity.

2.10 Continuous Integration

Continuous integration is the practise of constantly committing your work as a programmer to the main project[6]. This is a practise that has become more and more popular as a development method, as the purpose is to integrate more often, and thereby encounter less bugs when trying to merge the work of multiple programmers. There are different ideas of how often one should be integrating for it to be continuous, but the consensus is that it should be at least a daily occurrence

(if not multiple times a day)[6].

When integrating new changes to the main project it is also a common practise to have a testing environment setup for the project, which will automatically run and look for bugs introduced with the newest commit. There are two ways, or rather times, which these tests can be run. Firstly, the tests might be run immediately after a new integration is complete, to provide instant feedback to the developer if their integration was a success or a failure. This is the ideal way to have tests run when doing continuous integration, however it does have one problem, since some testing can take a long time to complete, as in the original case for Francisco G de Oliveira Neto et al. in [18]. If testing the system takes a while (hours) it will not provide the developer with instant feedback, which is the point of running them right away, and as such leads to the second way of running these tests. The second way for performing tests on a system is over-night tests, and while this is not ideal, it can be necessary. In [18] Francisco G de Oliveira Neto et al. look at optimising the runtime of their tests run with each commit, while still running the full test setup every night, which provides a good middle ground between instant feedback for the developer and performing rigorous and thorough testing.

2.11 Performance testing in continuous integration environments

In [7] Geiger et al. investigate how to conduct performance testing in a continuous integration environment, as opposed to only performing unit tests in such an environment. The way they setup the performance tests is through the use of a continuous integration server which is connected to the version control used in the development environment. When the CI server is made aware of a new version being available it builds the new version and performs the associated tests on the new build.

In the report Geiger et al. outline a number of different test types which fall under performance testing, these being; load testing, stress testing, capacity testing, component testing and finally unit testing.

Furthermore, the report describes what different performance regression tools are available in Java along with what different CI services are available. The authors then try and integrate the found tools into the CI services and evaluate on nine criteria. These nine criteria are given between 0 and 2 points, with higher points being better. Not all found tools are available for each of the different CI services and the report only attempts to integrate the tools if there is a direct plugin for the given tool to the CI service. The report concludes that each of the different CI services has support for some performance regression testing.

In section 2.8 it is described how Sommerville defined 3 types of testing, namely; unit test, component test and system tests. This is in line with what Geiger et al. outline, as load, stress and capacity testing could all be seen as system tests.

2.12 Considerations for Performance testing

The goal when using performance tests is to prevent performance regressions. This is how Andrey Akinshin puts it in [1]. However, analysis of the results of these tests is required to know what they are actually telling you. Akinshin underlines the importance of performance analysis in relation to the performance testing, as analysis of performance metrics is required to utilise the testing results.

Akinshin presents six goals for performance testing[1, pg.265], which are what developers should strive for when designing and analysing performance tests. These six goals are:

1. Prevent Performance Degradations
2. Detect Not-Prevented Degradations
3. Detect Other Kinds of Performance Anomalies
4. Reduce Type I Error Rate (False positives)
5. Reduce Type II Error Rate (False negatives)
6. Automate Everything

1. Through an example he stresses how performance regressions have different impacts depending on where in a code base they occur. And he explains how a 500% increase on a method which is rarely run has a much smaller impact than a 1% increase of a frequently run method.

2. Akinshin also describes how the purpose of performance testing is not only to prevent performance degradations, but also to detect when non-preventable degradations occur. This is because not all degradations are preventable, but if the programmers are aware that a degradation has occurred, they can then investigate the importance of the degradation and act accordingly to fix it.

3. Given a benchmark, there may be other reasons for a degradation in performance. Akinshin classifies these "performance anomalies", which are clustering of data, large variance or other strange performance distributions. These problems can be caused by errors in the business logic of the program.

4. + 5. Another thing Akinshin warns about is both false negatives and false positives. He proposes reducing performance requirements if too many false positives are encountered, as dealing with multiple cases of false positives per day

developers will start to not care about them, in a "The boy who cried wolf" scenario. Additionally if false negatives occur, this means that the original goal of detecting performance regressions is not met to the extend that would be desired.

6. Finally, Akinshin suggests that developers automate as much of their performance pipeline as possible, from the tests to the analysis of the performance metrics. This is to reduce the human element, and avoid having one person handling performance, and in turn depending on that person for the performance to not degrade. He does however note that automating everything is not always feasible, but it should be a goal nonetheless.

2.12.1 How this relates to energy measurements

Performance and energy metrics are quite similar when it comes to measuring, as most performance metrics are done through runtime, and energy measurements are closely related to the runtime. So closely related that many energy models use energy per time as their measurement for power consumption. And similarly for Edora as described in section 1.1, where they use total energy usage of the server per transaction. While performance testing focuses both on prevention as well as detection of degradations, the energy measurements will only focus on detection, as the energy measurements will be performed as a wrapper for the tests. This means that the tests themselves are not testing the energy usage, and as such can not prevent potential energy degradations, but will be made to serve as a heads up for programmers, that there has been a significant energy usage increase.

2.13 Distributions

When performing performance tests, regardless of if its time or energy based, the result of those tests are not a pass or failed as with other tests in programming. Instead, the tests are run multiple times and a distribution of results are found. If performance tests were only run once, the results could be misguiding if there was anything impacting them, such as JIT compilations or background daemons using CPU cycles[1, pg.148], which was also a described in section 2.3. In turn, if these tests are run multiple times, these factors will have less of an impact. These distributions can take different forms depending on which type of software or hardware component is being tested. The most common distribution is a normal distribution which will be described in subsection 2.13.1. Another type of distribution is a bimodal or multimodal distribution, which is presented in subsection 2.13.2. These are only a few of the many types of distributions which exist. In subsection 2.13.3 the skewness of distributions are described.

2.13.1 Normal Distribution

A normal distribution, or Gaussian distribution, is used in statistics to describe a default probability distribution. It describes how a regular behaviour of the measured statistic can still have deviations or randomness involved without being false or incorrect measurements[21].

A normal distribution has a probability density function which can be described as Equation 2.1.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} \tag{2.1}$$

Where μ is the mean, σ is the standard deviation, and x is the variate.

A normal distribution with such a probability density will result in what is called a "bell cruve" graph, which can be seen on Figure 2.2.

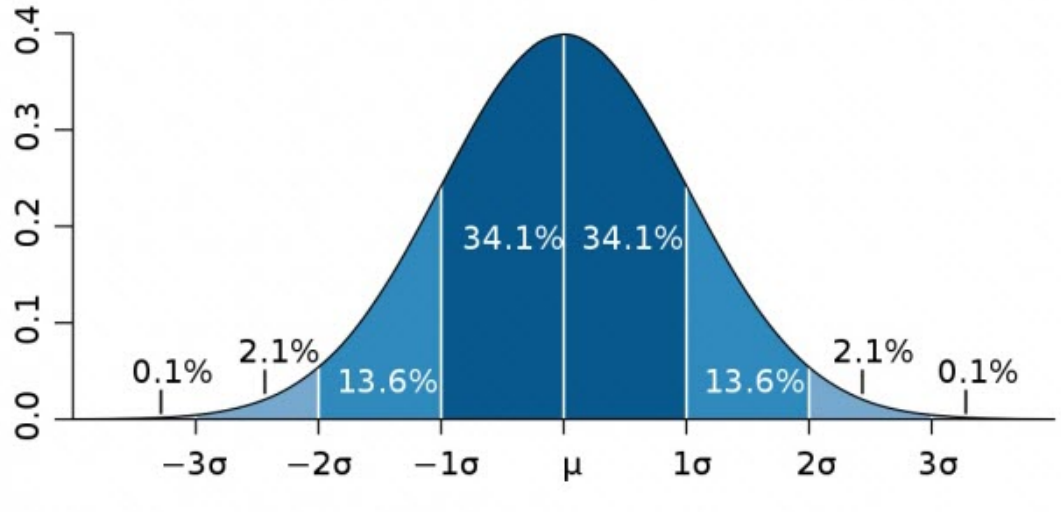

Figure 2.2: Bell Curve with standard deviations [5]

Here the results are concentrated around the mean, and the probability of outliers drastically falls off with each standard deviation from the mean, as denoted by the vertical lines.

2.13.2 Multimodal distribution

A multimodal distribution is similar to a normal distribution, the difference comes from the number of peaks in the distribution[1, pg.314]. While a normal distribution has just one peak a multimodal distribution can have two or more peaks. A distribution with exactly two peaks is also called a bimodial distribution.

If the peaks in a bimodial distribution are uneven, the highest peak is called the major mode and the lower peak is called the minor mode.

2.13.3 Skewness

Skewness in a distribution is when the results have a tail to either side of the main distribution[1, pg.204]. A right tail is a positive skewness and a left tail is a negative skewnss. A perfect normal distribution will as such have no tail and thereby no skewness. Skewness can be a result of large standard deviations, but in software these deviations can be caused by a wide array of known or unknown issues, as touched upon in the start of this section.

2.14 Mann-Whitney U test

To assure that the found results are significant, and not occurring by chance, a Mann-Whitney U test is performed. This test is chosen over a Student's t-test or Welch's t-test as it is more versatile for different distributions[1, p.222]. It was previously outlined in subsection 2.13.1 that different types of distributions can be encountered. The purpose is to find the p-value, which is the probability that these distributions are from the same population, and thereby not different. The threshold for the p-value is set at 0.05 which is the standard threshold for indicating statistical significance. First, a null hypothesis is made, as well as an alternative hypothesis:

H_0 = The two distributions are equal

H_1 = The two distributions are not equal

A two sided alternative hypothesis is used, as opposed to a single sided one. This is because the interest lies is determining if there is a difference or not, and not which way the potential difference lies. Should there be a statistically significant difference, then this difference will be visually shown through graphs.

If the p-value for a distribution comparison is below the 0.05 threshold the null hypothesis can be rejected and the alternative hypothesis is accepted.

2.15 The p-value problem

When performing statistical significance tests, a problem which can be encountered is that the p-value approaches 0. This problem is presented by Galit Schmueli et al. in the paper "Too Big to Fail: Large samples and the p-Value Problem"[12]. The p-value, or probability value, is the measurement which is used to indicate the probability that two different samples could be from the same distribution. The p-value approaches 0 when the sample size gets larger, but there does not exist a specific cut of point for when a sample size is considered too large and

the p-value is no longer relevant. In the paper they find that the problem can start from about 200 samples but also use sample sizes of 10 000 to indicate a large sample. This problem needs to be kept in mind before declaring results statistically significant solely based on p-value size. Additionally, the authors suggest that an alternative to simply presenting p-values can be to visually present the data as well as commenting on the effect size.

Chapter 3

Initial RAPL experimentation

Before we are able to do any measurements with RAPL in automated tests, we first need to implement RAPL measurements in tests. To extend from our previous work in section 2.1, `rapl.rs` was originally used to see if it was possible to apply its functionality for testing. However when comparing an execution of a test in a unit testing framework versus running the same function from command-line, it was discovered that execution times for the unit test were longer. As such we concluded that in order to measure the tests using RAPL values, we need to integrate RAPL further into the testing framework, similar to how [17] integrated their energy readings into the code, which was presented in section 2.2.

3.1 Test Framework

Given the motivation in section 1.1, the programming language which was chosen is C#. For C# there exists a number of test frameworks for conducting your tests. It only makes sense to pick one testing framework, for this project. Of the most popular options, there are three major frameworks. These are MSTest V2[1], xUnit.net[2] and NUnit[3]. These three popular frameworks all have in common, that they are extendable with custom testing methods. This means that these three all seem like good candidates for extending with RAPL functionality. As MSTest V2 appears to be the official testing framework by Microsoft, it was chosen as the most clear candidate for implementing RAPL functionality into.

[1] `https://github.com/Microsoft/testfx`

[2] `https://xunit.net/`

[3] `https://nunit.org/`

3.2 Measuring time & energy

Energy measurements on their own can only tell if a section of code consumes more or less energy compared to a different version of the code. This can be useful if optimizing the code for less energy consumption. The process of collecting energy consumption is to record the energy counter before and after executing the code. Similarly, the process of collecting the time consumption is to record the clock before and after executing the code. Given how relatable the methods of doing energy measurements and time measurements are, also collecting the time consumption of a piece of code is straight forward. This gives the data analysis another dimension to consider, and since the tests are performed anyway with energy measurements, having the time recorded as well comes at very little cost. Having both measurements is helpful since time and energy improvements are not always related[3].

Next comes the question of how to measure time accurately. [1] has a thorough explanation of this topic. In the book, three different timer approaches are mentioned. The options being the following: `DateTime.UtcNow`, `Environment.TickCount` and `Stopwatch.GetTimestamp`. On the system that [1] used, `DateTime.UtcNow` has a resolution of 100 ns and a latency of 26-30 ns while `Environment.TickCount` had a resolution of 3.9-4.0 ms and a latency of 8-10 ns. `Stopwatch.GetTimestamp` a resolution of 30-40 ns with a latency had a latency of 70-80 ns. It is concluded that `Stopwatch` is the best solution when high-precision timestamping is required. This is because resolution is the most important aspect in this case.

3.3 Experimenting with a RAPL implementation in MSTest V2

In Appendix A an implementation for RAPL in MSTest is shown. In this code we extend the test attributes in order to incorporate our own methods into the testing, which in this case allows us to run the RAPL measurements directly within the test framework, which results in less overhead for the measurements, compared to an external wrapper.

Additionally, we override the `Execute` method from the test framework, so that it is able to run the tests as many times as needed, as opposed to a singular test run. The number of times the test is run depends on the number provided along with the call to the test method.

It should be noted that to run the tests using RAPL in this way requires the user who executes the tests to have read permissions for the RAPL-counters, which is normally restricted to only the root user or group. This can be changed by changing the permissions for the RAPL-counter to allow every user access.

3.4 Build configuration

.NET has support for different build configurations[15]. The most common are *Debug* and *Release*. The major difference in *Debug*- and *Release*-configuration is that the *Debug*-configuration enables debugging information and compiles without optimizations [14]. For benchmarking, [1, pg.37] says to never use the debug build. This seems intuitive as the debug mode performs extra tasks in order to create debug information and as a result runs slower compared to an optimised build. The experiments performed in this report will be done using the release configuration, unless otherwise specified.

3.5 Test setup

The hardware specifications of the computer are as follows:

- Computer type: Optiplex 5050 (07A2)
- Motherboard: Dell 0WWJRX
- RAM: 2x 8GB; 2400 MHz; DDR4; DIMM; Micron Technology (8ATF1G64AZ-2G3B1) and Hynix Semiconductor (HMA81GU6AFR8N-UH)
- CPU: Intel i7-6700; Skylake; 3.4GHz (overclock: 4.2GHz); 64bit; clock 100MHz
- GPU: integrated
- Disk: 256GB SSD; INTEL SSDSC2FK25

The software specifications of the system is as follows:

- Ubuntu Server 21.10, minimal install
- lightDM as login manager
- i3 as window manager (and as such, the systems runs X11)

Linux is chosen to be able to utilize RAPL.

For the programming languages requiring a runtime, the runtimes are as follows:

- C#: .NET 6.0.200

3.6 Sample program for testing

An initial unit test experiment is conducted, to attempt to relate the energy consumption of a test, to the energy consumption of the program under test. A simple Fibonacci calculation is chosen as the initial experiment. This simple Fibonacci calculation test case was to be a proof of concept that the energy of a test can be measured. In Snippet 3.6.1 the code for the initial implementation of a fibonacchi function can be seen.

For executing the program, the `main` function can be seen in Snippet 3.6.2.

```
public static int GetFib(int n) {
  int number = n;
  int[] Fib = new int[number + 1];
  Fib[0] = 0;
  Fib[1] = 1;
  for(int i = 2; i <= number; i++){
    Fib[i] = Fib[i - 2] + Fib[i - 1];
  }
  return Fib[number];
}
```

Snippet 3.6.1: Implementation of `GetFib` with a complexity of $O(n)$

```
using UtilityLibraries;

class Program {
  private const string FILE_PATH =
  ↪ "/sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/energy_uj";

  private static Decimal read_rapl_value() {
    string raw_value = System.IO.File.ReadAllText(FILE_PATH);

    return Decimal.Parse(raw_value);
  }

  static void Main(string[] args) {
    Console.WriteLine("Iteration;RAPL-Value;Old-RAPL;New-RAPL;Ticks");
    for(int count = 0; count < 25; count++){
      Decimal before_value = read_rapl_value();
      var watch = System.Diagnostics.Stopwatch.StartNew();

      Console.WriteLine(FibLibrary.SlowGetFib(42));

      Decimal new_value = read_rapl_value();
      watch.Stop();
      Console.WriteLine("{0};{1};{2};{3};{4}", count, new_value-before_value,
      ↪ before_value, new_value, watch.ElapsedTicks);
    }
  }
}
```

Snippet 3.6.2: Main-program representing how the MSTest would be implemented as a main.

3.7 Initial test results

In Appendix B the results of executing the tests can be seen. As is obvious from the results , the RAPL-value is often 0. If we now focus on the number of ticks (Which is C#'s StopWatch Ticks[4]), we can see that the number is very often around $\approx 30\,000$ ticks. Ticks on the experiment system is equal to nano seconds ($10^{-9}s$). That means that the time elapsed between samples is often around 30 μs. According to [13], the sampling rate of Intel RAPL counters were documented to be every 1 ms, but they accomplished a sampling rate of $\approx 50\mu s$. This was explained in section 2.6.

[4] https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch.elapsedticks?view=net-6.0

The results are unexpected when considering the way of measuring these functions highly resembles they way [17] did their measurements. In [17], they used this way to establish their "Ground truth" measurements. Unfortunately, they do not mention if their results were also in the micro-seconds range and if their results also often measure 0 joules consumed.

From this result we need more research into how the measure a function like this Fibonacci implementation. In the following section we will look into a slower implementation of a Fibonacci sequence.

3.8 Slow fibonacchi

In order to test if energy usage is measurable at all, with the current implementation, a longer running implementation of a fibonacchi sequence function is tested. This implementation can be seen in Snippet 3.8.1. Where the previous solution had a time complexity of $O(n)$, this slower version has a time complexity of $O(2^n)$.

```
1 public static int SlowGetFib(int n) {
2   if(n == 1)
3     return 1;
4   if(n <= 0)
5     return 0;
6   return SlowGetFib(n - 2) + SlowGetFib(n - 1);
7 }
```

Snippet 3.8.1: Implementation of `SlowGetFib` with a complexity of $O(2^n)$

The results of the `GetSlowFib` experiment can be seen in Appendix C, here it is seen how the calculations are not reduced to 0, like in the `GetFib` experiment. The results here are quite similar between each iteration, with the exception of the first two or three runs which could be attributed to the just-in-time compiler, which sometimes makes the first uses of a method take longer to run, as it performs the compilation during those first usages.

From this result, as well as the results from the faster Fibonacci experiment in section 3.7, it is evident that more research into the capabilities and limitations of Intel RAPL is necessary. In the following section we will examine some of these limitations and create a more robust method for doing energy measurements with Intel RAPL.

3.9 Intel RAPL update rate

Platypus mentions an update rate which is higher than the documented update rate from Intel. Intel mentions an update rate of ≈1 millisecond[8, pg.535], while Platypus[13] mentions an update rate between 0.05 milliseconds and 1 millisecond, depending on which domain is measured, regardless of being accessed via driver or kernel.

This raises the question:

What is the sampling rate of Intel RAPL?

In section 2.7, we mentioned how Intel had handled the Platypus attack's impact by reducing the accuracy and sampling rate of Intel RAPL. Intel also mention how to disable the features, by disabling Intel SGX. Controlling the status of Intel SGX is a feature in the BIOS of the test system. Intel provides a tool for checking the SGX status[5].

For testing the sampling rate, it would be convenient if there was an existing mechanism for it. In fact, there exists a tool called `inotifywait`[6] which can look for file changes. This tools use-case is to act on changes to a file. Developers may know this from when you save a file and want the computer to rebuild the solution with the change. Unfortunately, this approach cannot be utilized since the underlying kernel functionality `inotify`[7] cannot monitor all of the pseudo-filesystem, including `/sys`, which is where RAPL is located. Another approach is to have an indefinite loop which reads the file and reports when the file is changed.

3.9.1 Number of iterations

When performing benchmarks, an important question is how many tests should be performed. In this case each iteration takes a millisecond or less to perform. Ideally as many iterations as possible are performed, in order to have the most representative data available. While large background processes can have an impact on the readings, these processes can occur at random times, and could require testing the system for days if not weeks in order to capture, these different processes were described in section 2.3. Changes to the sampling rate are also not expected, as this update should happen on the CPU regardless of background processes. If the sampling rate is run for one hour, the resulting data will be at a minimum 3.6 million samples, according to the documented 1ms update rate. This is deemed sufficient and therefor one hour run time is chosen.

[5] `https://github.com/intel/sgx-software-enable`

[6] `https://man7.org/linux/man-pages/man1/inotifywait.1.html`

[7] `https://man7.org/linux/man-pages/man7/inotify.7.html`

3.9.2 Update rate experiment

```
private static void rapl_update_interval(int measurement_time_secs) {
  List<long> measurements = new List<long>(1_000_000_000);
  var outer_watch = Stopwatch.StartNew();

  while (outer_watch.ElapsedMilliseconds < measurement_time_secs * 1000) {
    bool changed = false;
    var inital_value = read_rapl_value();
    var watch = System.Diagnostics.Stopwatch.StartNew();
    while (!changed) {
      var new_value = read_rapl_value();
      changed = new_value != inital_value;
    }
    watch.Stop();
    measurements.Add(watch.ElapsedTicks);
  }
  System.Console.WriteLine(String.Join("\n", measurements));
}
```

Snippet 3.9.1: Main-program to measure sampling rate of doing Intel RAPL measurements

With the code in Snippet 3.9.1, the sampling rate for the core and package domains was tested. The code uses a busy-waiting loop in order to detect the changes in RAPL values. The results of the one hour runs can be seen in Figure 3.1 and Figure 3.2. From these histograms it can be seen that SGX does not have an impact on the update rate of RAPL.

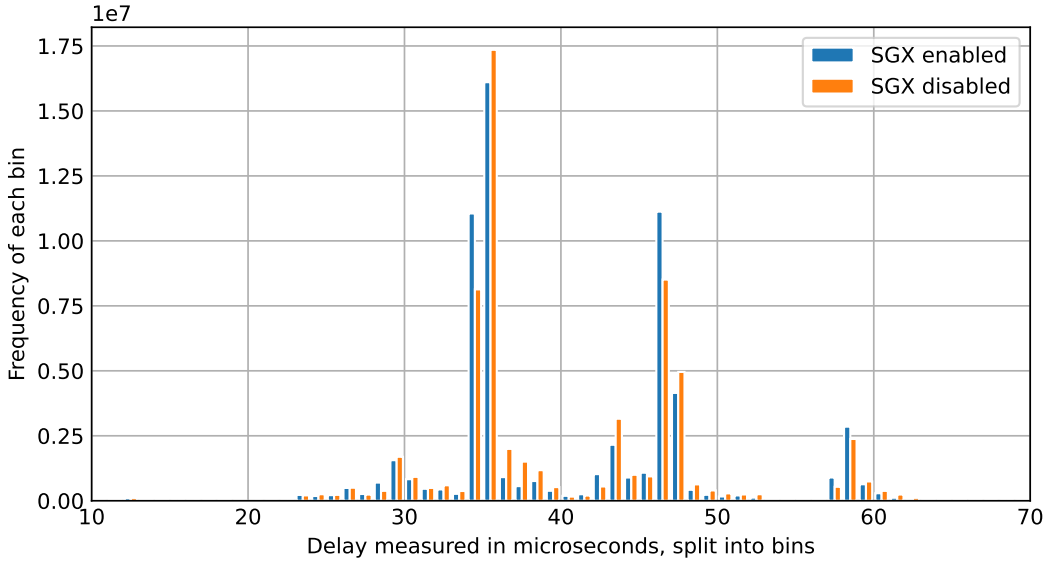

Figure 3.1: Histogram of the sampling rate of Intel RAPL using busy waiting over 1 hour. $n_{sgx\,enabled} = 63\,770\,108, n_{sgx\,disabled} = 63\,779\,838$ measuring the core domain, p-value = 0.0

For the sampling rate test of the core domain, the minimum and maximum values, as well as the sampling rate percentiles can be seen in Table 3.1. Where the run times are at or below the threshold in the table, for the given percentile for both SGX enabled and disabled.

Core	**Minimum**	**95 percentile**	**99 percentile**	**Maximum**
SGX enabled	8.21 μs	58.53 μs	114.15 μs	5 411.75 μs
SGX disabled	8.22 μs	58.67 μs	114.1 μs	5 322.13 μs

Table 3.1: Sampling rate percentiles for an hour run time with core domain measurements

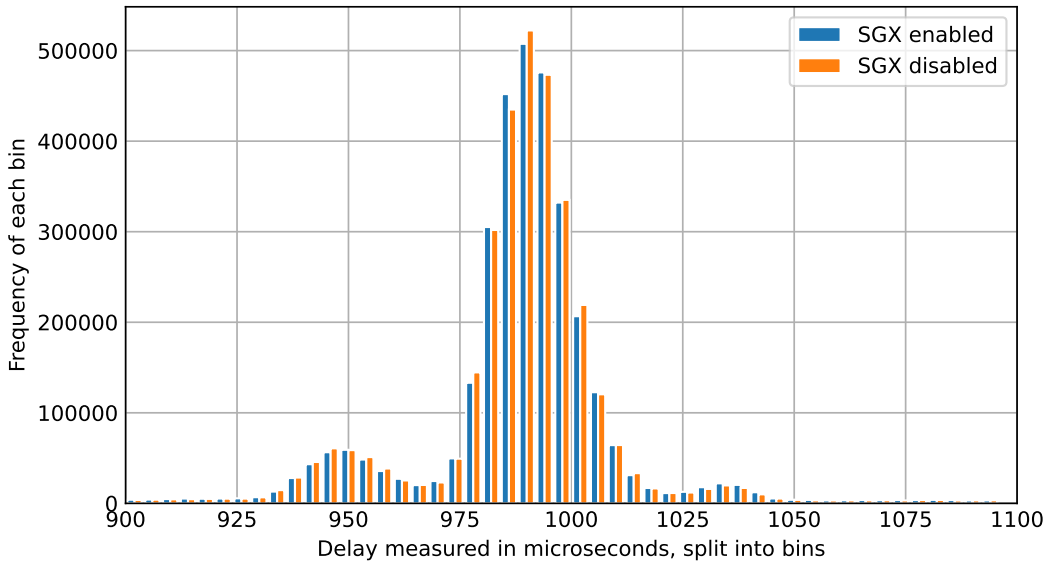

Figure 3.2: Histogram of the sampling rate of Intel RAPL using busy waiting over 1 hour. $n_{sgx\,enabled} = 3\,600\,009, n_{sgx\,disabled} = 3\,599\,945$ measuring the package domain, p-value = 7.50e-11

Package	**Minimum**	**95 percentile**	**99 percentile**	**Maximum**
SGX enabled	81.78 μs	1077.83 μs	1395.84 μs	6 207.63 μs
SGX disabled	21.9 μs	1068.97 μs	1384.42 μs	6 628.15 μs

Table 3.2: Sampling rate percentiles for an hour run time with package domain measurements

During our sampling of the test computer it is evident that the sampling rate on the core domain is higher than the documented 1ms on the system used. As can be seen in Table 3.2, we see that the 95th percentile of the package domain is 1077.83μs for SGX enabled and 1068.97μs for SGX disabled, which is close to the documented value.

When comparing the results from the two different 1 hour sample rate runs shown in Figure 3.1 and Figure 3.2, it is clear that there is a big difference if the package or the core is sampled. In both cases the distributions are multimodal, with a clear mode major and mode minors, as described in subsection 2.13.2.

The found results are in line with what is described by Platypus, and assuming that Intel is talking purely about the package sampling, they are also inline with the documentation. However, the fact that the core sampling is faster than the package allows for more accurate measurements, especially when working with micro benchmarks. Only measuring the core can mean missing changes in the energy usage of DRAM or uncore. With regards to differences between SGX disabled and SGX enabled, a very low p-value can be observed. However when looking at

the plot and the percentiles, it is clear that SGX status does not have a major impact on the sampling rate of Intel RAPL.

To conclude, it is important to record the sampling rate of the system used for testing in order to know what the capabilities of the test system are, as well as considering if the core or package domain should be sampled. In this case, the higher-than-documented sampling rate may provide more accurate energy measurements of the system when measuring on smaller functions.

3.10 C# Loop rate

After determining the sampling rate of Intel RAPL, the next step is questioning if there is certainty that this is the sampling rate of Intel RAPL and not the time for each iteration of code. This raises the following experiment question:

Are we able to capture all Intel RAPL updates using C# code?

To test this experiment, we write C# code which iterates as fast as possible, while noting the time and the RAPL value. Given the output of the data we will be able to answer the question above. If the RAPL value changes on every iteration, it would mean that Intel RAPL updates faster than the code is able to iterate. In such a case we would unable to capture all Intel RAPL updates using C# code. Instead, if RAPL values remains identical on multiple iterations, it would mean that the iterations are faster than Intel RAPL updates. In this case, we are able to capture all Intel RAPL updates using C# code.

In Snippet 3.10.1 the code used to test the above mentioned experiment can be seen. The code loops as fast as possible (line 5-8), adding only the result to measurements. The measurements variable is declared as a List which is a dynamic list (line 2). This dynamic list is initialized with a capacity of 1 billion elements in order to minimize the number of resize operations.

```
private static void log_while_true_rapl(int measurement_time_secs) {
  List<(string, long)> measurements = new List<(string, long)>(1_000_000_000);
  var watch = Stopwatch.StartNew();

  while (watch.ElapsedMilliseconds < measurement_time_secs * 1000) {
    var rapl_value = read_rapl_value();
    measurements.Add((rapl_value, watch.ElapsedTicks));
  }
  foreach (var tuple in measurements) {
    System.Console.WriteLine(tuple.Item1.Replace("\n", String.Empty) + ";" +
    ↪ tuple.Item2);
  }
}
```

Snippet 3.10.1: Function to sample how often C# is able to loop

For the number of iterations, the same approach is used as in subsection 3.9.1. That is one hour of samples and if our estimation of observing every update of RAPL is accurate, the resulting data will be ≈ 60 million samples. This is because we are examining the core domain, where the sampling rate was found to be every $60\mu s$. While Intel SGX made no impact in the previous experiment, this experiment is also conducted with both Intel SGX disabled and enabled. This is to test if there is a difference in the frequency at which the RAPL file is changed, between SGX enabled and disabled. The data produced by the sampling is a long list of RAPL values and the time. The number of identical RAPL values are counted and plotted in Figure 3.3. An important number for this plot is that for SGX enabled, 55.4 million changes were counted. Of these 52.7 million were above 3 iterations. For SGX disabled, 53.2 million changes out of 56.3 million changes were above 3 iterations. So the results are highly similar. Furthermore, for comparing SGX enabled vs SGX disabled, we see a bit of a difference with number of iterations completed. We assume that the run time of the C# code is constant since it is the same code. When SGX is enabled, the results appears to be a bit skewed towards 5-6 loops on every RAPL update, which could indicate that RAPL updates a bit slower. However when compared to the previous experiment, the change was minuscule. As such, SGX enabled or disabled does not seem to have a major impact on the results.

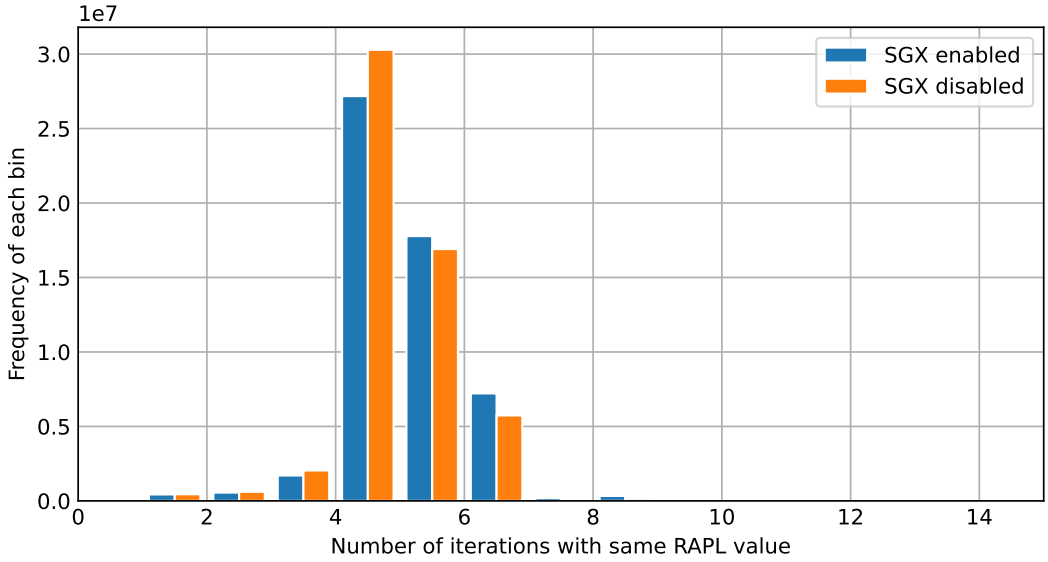

Figure 3.3: Loop rate of a benchmark over 1 hour. $n_{sgx\ enabled} = 251\,631\,154, n_{sgx\ disabled} = 249\,737\,854$ measuring the core domain, p-value = 0.0

In this experiment we tested if C# code is able to capture all Intel RAPL updates. From the data presented in Figure 3.3, we can conclude that C# code is able to capture all RAPL updates for this specific system. This test was performed with measuring the core domain, and since the package domain updates less frequently, no measurements would be missed on the package domain either.

The processes of testing the sample rate of Intel RAPL and the loop rate of C# are also available in a single repository[8] This is to enable others to replicate the experiment on their system.

3.11 Benchmarking with a sampling rate

The sampling rate of Intel RAPL has a significant impact on the measurements that can be done. This means that the sampling rate needs to be accounted for. In Figure 3.4 a graphical representation can be seen of how a measurement is conducted. The horizontal line is the time, above the line is how Intel RAPL works and below the line is how the benchmark is run. Intel RAPL has a certain sampling rate, which is out of control of the user. If the sampling rate of Intel RAPL is not taken into account, the beginning and end of a benchmark will fall in between an Intel RAPL update. The best case of measurements is if the beginning and end of the benchmark is synchronized with a RAPL update. The worst case of

[8] `https://github.com/energyci/rapl-limit-tests`

measurements is if both the beginning and end of the benchmark is in between a RAPL update. This means that the benchmark vary from two bad samples, in the worst case, to zero bad samples in the best case.

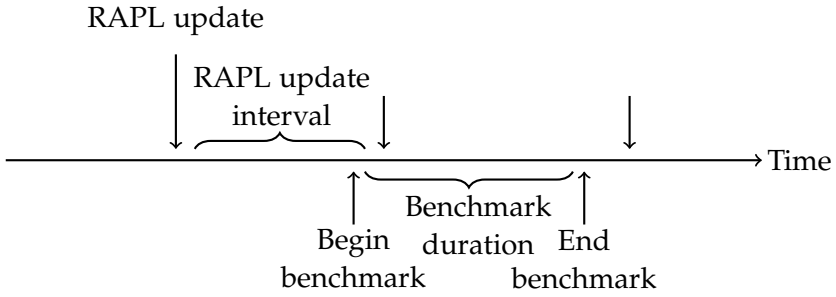

Figure 3.4: Sampling rate during a benchmark

3.12 Minimum measurable test

In section 3.11, we noted that from a benchmark, there was a potential for up to two bad samples in the worst case. With the sample rate found, it is possible to calculate the minimum measurable test to a certain extent using Equation 3.1.

$$t_{duration} = \frac{2 * samplerate}{1 - confidence} \tag{3.1}$$

Using the numbers from the determined sampling rate in the equation, the following numbers are found:

$$t_{core} = \frac{2 * 60\mu s}{1 - 0.95} = 2400\mu s \quad t_{package} = \frac{2 * 1\,080\mu s}{1 - 0.95} = 43\,200\mu s$$

For the 95 percentile of sampling rate from the core domain, and a confidence of 95%, the result is $2400\mu s$ or 2.4 ms. For the package domain the same percentile with the same confidence, is 43.2 ms. If the test is able to run for longer, the two bad samples have less of an impact.

3.13 Test time

In subsection 3.9.1 the number of iterations for an experiment was discussed. In that section the use case was in regard to experiments performed on the system, to specify sampling and loop rates. In this section the experiments are in relation to running the tests within a test framework a number of times to active information on the energy usage of the system. As such there are different aspects of the development flow which needs to be considered.

If the test suite is to be run once a day, or once a week the suite itself as well as the number of iterations can be much larger than if it is included in the continuous integration framework, where it might be run multiple times a day. This was discussed further in section 2.10. In his article regarding continuous integration[6] Martin Fowler states that a 10 minute build time is "perfectly within reason" and also in line with the XP (eXtreme Programming) guidelines.

In order to attempt to fit the energy testing within the continuous integration, it is decided to run the test suite for 10 minutes. In actual production this will need to be reduced further, seeing as other parts of the build take up part of that 10 minute time frame as well. It should also be noted that as the energy test suite grows, the number of iterations which can be performed within this 10 minute time frame will reduce.

3.13.1 Time based testing in MSTest V2

In Snippet 3.13.1, the time based version can be seen.

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
public class IterativeTestMethodAttribute : TestMethodAttribute {
  private int stabilityThreshold;
  private const string FILE_PATH =
  ↪ "/sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/intel-rapl:0:0/energy_uj";

  private string read_rapl_value() {
    return System.IO.File.ReadAllText(FILE_PATH);
  }

  public IterativeTestMethodAttribute(int stabilityThreshold) {
    this.stabilityThreshold = stabilityThreshold;
  }

  public override TestResult[] Execute(ITestMethod testMethod) {
    var results = new List<TestResult>();
    var tuples = new List<(decimal, long)>();
    var begin_watch = System.Diagnostics.Stopwatch.StartNew();
    TestResult[]? currentResults = null;

    while (begin_watch.ElapsedMilliseconds < stabilityThreshold * 1_000) {
      string before_value = read_rapl_value();
      long before_time = begin_watch.ElapsedTicks;

      currentResults = base.Execute(testMethod);

      string new_value = read_rapl_value();
      long after_time = begin_watch.ElapsedTicks;

      var energy_consumption = Decimal.Parse(new_value) -
      ↪ Decimal.Parse(before_value);
      // Time in ticks.
      var time_elapsed = after_time - before_time;
      tuples.Add((energy_consumption, time_elapsed));

    }

    if(currentResults != null)
      results.AddRange(currentResults);

    foreach (var tuple in tuples) {
      System.Console.WriteLine(tuple.Item1 + ";" + tuple.Item2);
    }
    return results.ToArray();
  }
}
```

Snippet 3.13.1: Time based Intel RAPL measurements

3.14 Summary

From the limited experiment of a Fibonacci-function, we can conclude that it is possible to create a test approach that enables energy testing through the use of tests. The energy consumption of the unit tests highly resembles the energy consumption of the production ready function. However one needs to be aware of the sampling rate of Intel RAPL in order to make qualified energy measurements. In the specific test system used in this report, the 95th percentile sampling rate of Intel RAPL for SGX disabled in the core domain is 58.67 microseconds. For the package domain, 95th percentile with SGX disabled, the sampling rate is 1068.97 microseconds. Additionally, no large difference was found between SGX enabled and disabled. This leads us to not investigating these two settings further, as it requires every experiment to be run twice.

Chapter 4

Newtonsoft: Integer serialization

In this chapter it is examined how energy tests can be integrated into a larger project. First it is discussed how to implement energy testing into the architecture of a larger system. Afterwards the specifics of how the tests are integrated into the Newtonsoft.JSON library is presented.

4.1 Architecture

In the book Pro .NET benchmarking, which was introduced in section 2.12 the author also describes which types of existing tests can be used for performance testing[1, pg.272].

From this section is it clear that there is a multitude of ways for approaching performance testing. When talking about unit and integration tests, the author describes two types of performance tests from these, namely explicit and implicit performance tests[1, pg.287]. The implicit tests are existing integration tests which are wrapped in a stopwatch method to measure time performance. Additionally, explicit performance tests are described as tests that are meant to specifically evaluate on the performance, and provide feedback as a conclusion on performance. This is opposed to the implicit performance test, which is written to provide feedback as a passed or failed, where the performance metric is execution time which is measured with the previously mentioned stopwatch functionality.

Another difference between explicit and implicit performance tests, which is specified in the book[1], is the number of times these tests are run. Explicit performance tests are described as run multiple times, and with the result being a distribution, as described in subsection 2.13.1. Implicit performance tests are usually only run once, to check for the previously mentioned pass or fail. This means that while implicit performance provide information regarding the performance, they can be very unreliable and even misguiding in the form of false positives or false negatives. Reducing both of these factors is part the six goals when designing

and analysing performance tests, as described in section 2.12.

These approaches extend to energy measurements and performance as well as for time-based performance. This is due to the processes of measuring time and energy being similar, as discussed in section 2.12.

With this perspective in mind, two approaches for the architecture are outlined as follows, divided into a full coverage approach and a specialized coverage approach.

4.1.1 Full coverage

A full coverage approach would incorporate both implicit and explicit performance tests into the test suite in an attempt to cover as many bases as possible. This would provide as much information about potential energy regressions within the system as possible. The downside of using a full coverage approach lies in the additional workload when setting up the tests as well as difficulty when comparing energy usage of different versions as the implicit performance test suite grows more frequently than the explicit one.

4.1.2 Specialized coverage

Specialized coverage focuses on explicit performance tests and is simpler for a developer to implement into their system. In addition, it also requires less attention to maintain, as the comparisons are always 1:1, unless new features are added specifically to the explicit performance testing suite. The downside for specialized coverage is of course that implicit tests are not covered, and this could potentially mean that an energy regression goes undetected.

4.1.3 Chosen architecture

Ideally the solution would support both full and specialized coverage, and as a result be as widely usable as possible, and leave the choice up to the given developer how they integrate the solution into their specific system. However, this is deemed to be outside the scope of this project. This leads to the decision of focusing on specialized coverage, and in turn explicit performance testing, as this is evaluated to be the simpler to implement of the two approaches.

4.2 Test setup of larger system

In order to find a larger library for C# to test, we look at the package manager NuGet. NuGet has download statistics for all of their published packages[1]. On

[1] `https://www.nuget.org/stats/packages`

this list, the library Newtonsoft.Json is the most downloaded. Newtonsoft.Json is a library for working with JSON in .NET. Additionally, Newtonsoft.Json has supported .NET Core, the version that runs on linux, since 2016. If we look at the release history[2] we find a changelog. An interesting release is release 11.0.1[3] because it contains release notes of improved performance. This release contains the following lines:

1. New feature - Improved performance when resolving serialization contracts by using ConcurrentDictionary
2. New feature - Improved performance of JToken.Path with a faster reverse
3. New feature - Improved performance of parsing Int32 JSON integer values
4. New feature - Improved performance of parsing and writing enum names

In order to test these changes, we first need to find the commit that implemented the change. We find the 3rd item, *Improved performance of parsing Int32 JSON integer values* in the following commit `https://github.com/JamesNK/Newtonsoft.Json/commit/3ea750d6d6465387d4f7ba22bc9821f35772cc48`. As the release notes suggest, the change improved the performance of writing Int32 values to a `JsonTextWriter`. The commit even contains a benchmark for analysing the performance of said feature. In order to test this change with regards to the setup we mentioned in chapter 3, we need to test the different versions. Before we can pick the versions that we want to test, we write a function which resembles the benchmark in the previously mentioned commit. The function can be seen in Snippet 4.2.1.

```
1  public static string SerializeIntegers() {
2    StringWriter sw = new StringWriter();
3    JsonTextWriter jsonTextWriter = new JsonTextWriter(sw);
4    for (int i = 0; i < 10000; i++) {
5      jsonTextWriter.WriteValue(i);
6    }
7    jsonTextWriter.Flush();
8
9    return sw.ToString();
10 }
```

Snippet 4.2.1: Function for serializing integers using `JsonTextWriter`

[2] `https://github.com/JamesNK/Newtonsoft.Json/releases`
[3] `https://github.com/JamesNK/Newtonsoft.Json/releases/tag/11.0.1`

4.2.1 Picking a version to test

For measuring the difference in energy consumption, we need to select two versions; one before the change and one after the change. Because the project is a library, the usual use case would be to pick between releases. In this case, that would be 10.0.3 and 11.0.1. This major version change however also includes other commits which could have an impact on the energy consumption of the code. The alternative is to measure on the change by comparing the energy consumption of the parent commit with the commit in question. This approach has the advantage of being certain that the change has to be due the code change. However this also requires a custom build of the library. For the purpose of correctly identifying the change for this commit, we choose the option to compare commits. However since a specific commit is necessary, we need to understand how the build process was for the repository at the time of the commit in question. For this repository, the project contains many different build targets. For this project only `netstandard2.0` is necessary, therefore the other targets are removed and the project compiled. The result is two different DLL's, one for each build. For the testing, the process is to swap the DLL file before running the test.

4.2.2 Test run time

Previously in subsection 3.9.1 and section 3.6 the subject of number of runs versus testing run time has been discussed. The goal of the energy testing suite is to integrate it into the continuous integration work flow, and as such ideally run the test suite every time there is a new build. In order to successfully have the energy testing suite running in this work flow, the run time needs to be 10 minutes, as previously discussed. As such this test is performed with a 10 minute run time, which as mentioned will be reduced further when additional tests are added to the suite.

4.2.3 Minimising the impact of noise

In [1, p.51] Andrey Akinshin describes how code in a benchmark should ideally be run for one second. However, he also states that in most cases 100 ms can be acceptable. This run time is to reduce the impact of noise on the measurements and it is done by running the code as many times as possible during the chosen time window and averaging the run time across the number of iterations performed. This is then done multiple times to find a distribution of averaged results. This applies to energy measurements in the same way as it does for time measurements. The minimum measurable run time for a test was presented in section 3.12. The range for run time can range from the calculated 2.4 ms (if measuring the core) to the one second advised by Andrey Akinshin.

While one second run time is described as the general recommendation, a 200 ms run time is chosen. This is due to the 10 minute overall testing time described in subsection 4.2.2. If one second run time was used, it would result in a maximum of 600 measurements total, even if running multiple benchmarks, which is deemed to be insufficient for creating a proper distribution for each benchmark. The 200 ms run time is chosen over 100 ms as a middle ground between noise reduction and total measurements in the time window.

The change of adding the averaging over a given amount of time can be seen in Snippet 4.2.2. The snippet shows a while loop which is nested within the loop shown in Snippet 3.13.1, line 22-line 25, additionally it also logs the number of iterations.

```
1  ...
2  long before_time = begin_watch.ElapsedTicks;
3  long iteration_time = begin_watch.ElapsedMilliseconds;
4  int iteration_counter = 0;
5
6  while (begin_watch.ElapsedMilliseconds - iteration_time < 200) {
7    currentResults = base.Execute(testMethod);
8    iteration_counter++;
9  }
10 ...
```

Snippet 4.2.2: Snippet for adding averaging over 200 ms

4.3 Results

The results of the Newtonsoft JSON test are presented in this section. When there is a reference to *Performance*, this term covers the performance commit from Newtonsoft.JSON, e.g. `3ea750d`. The term *Normal* references to the commit before the performance commit, e.g `a8245a5`.

As mentioned in subsection 4.2.3, the measurements are averaged over 200 ms. This means that the energy and time counters are recorded at the start of a measurement. Then the test method is run repeatedly for 200 ms where the number of times it is run is counted. Afterwards the energy and time counters are recorded, and the difference as well as number of runs is logged. The first 10 runs which equals 2 seconds of test time for the normal commit test is shown in Table 4.1.

These energy and time measurements are then divided by the number of runs to find the average energy cost and run time for that given 200 ms interval. These values are plotted in their respective graphs in the following subsections.

Core (μJ)	Package (μJ)	Time (ns)	Number of runs
3 403 617	3 713 065	200 243 592	573
3 303 153	3 611 685	200 401 899	675
3 484 855	3 804 861	199 674 845	673
3 297 721	3 609 427	200 285 581	679
3 304 496	3 628 347	199 533 228	677
3 276 054	3 590 079	200 046 008	679
3 498 770	3 813 162	199 905 797	668
3 555 471	3 875 722	199 846 783	592
6 087 753	6 413 009	199 724 825	641
4 297 535	4 621 875	199 922 860	659

Table 4.1: The first 10 measurements from Newtonsoft.JSON test for the normal commit (a8245a5)

4.3.1 Average energy per iteration

The energy usages are plotted for each iteration in Figure 4.1 and Figure 4.3, which are core and package domains respectively. Here each data point is the average energy usage for that 200 ms interval. It is clear from the graphs that there is a difference between the normal and performance commit. However, it is not clear how that difference varies over the 3000 iterations. The difference between the two commits energy usage per iteration is shown in Figure 4.2 and Figure 4.4.

The first samples have a higher average energy consumption compared to the rest of the results. For C# this is potentially the Just-in-time compiler doing some initialization and optimization. An interesting trend in the figure is that the per iteration energy consumption is trending upwards for the first $\approx$ 1000 measurements. There are also occasional outliers.

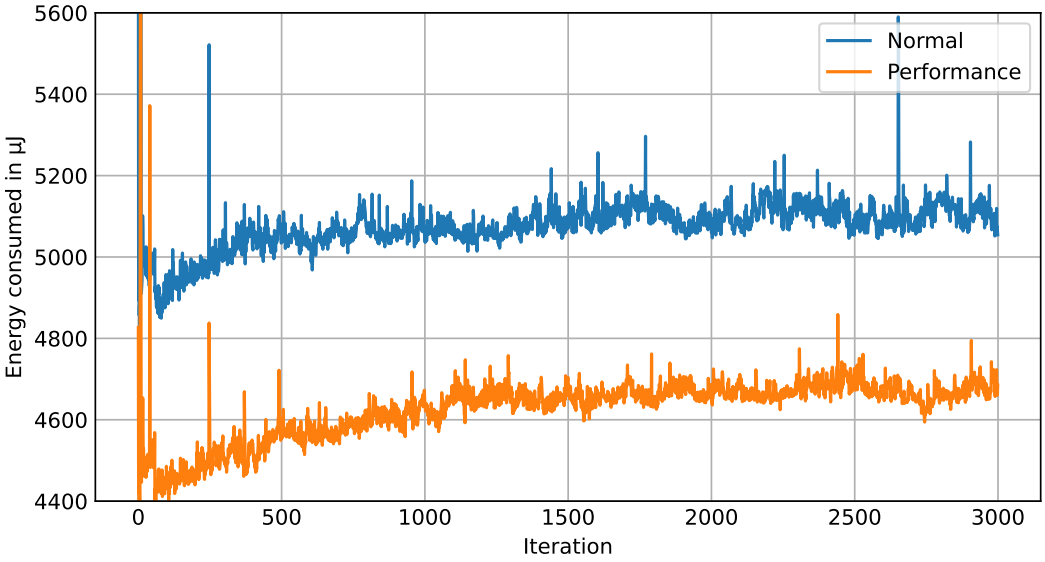

Figure 4.1: Average energy consumed for Newtonsoft.JSON test on the core domain, with total number of runs $m_{performance} = 2\,265\,471$, $m_{normal} = 2\,044\,789$, p-value = 0.0

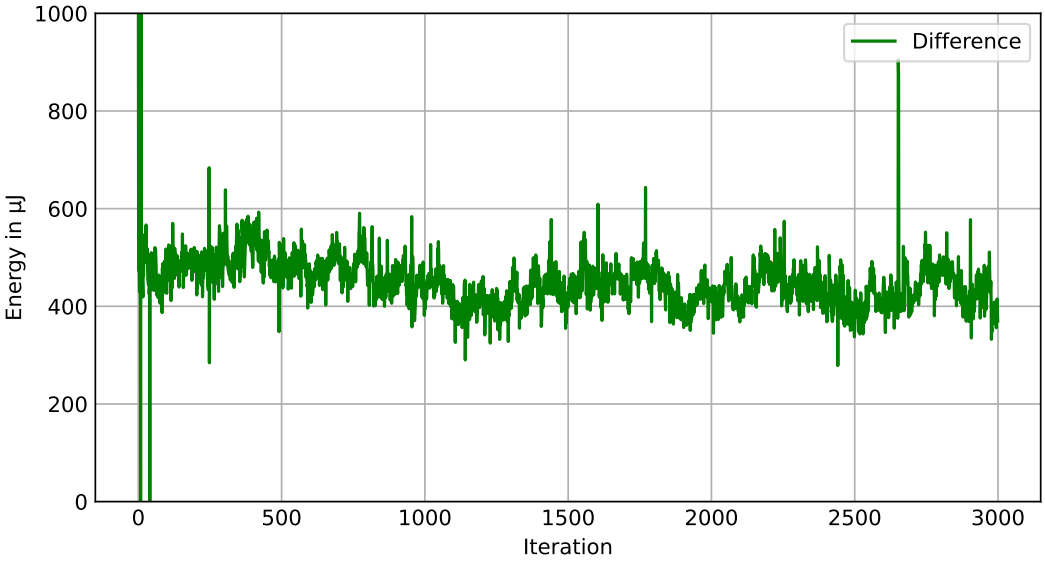

Figure 4.2: Difference in energy consumed for the core domain

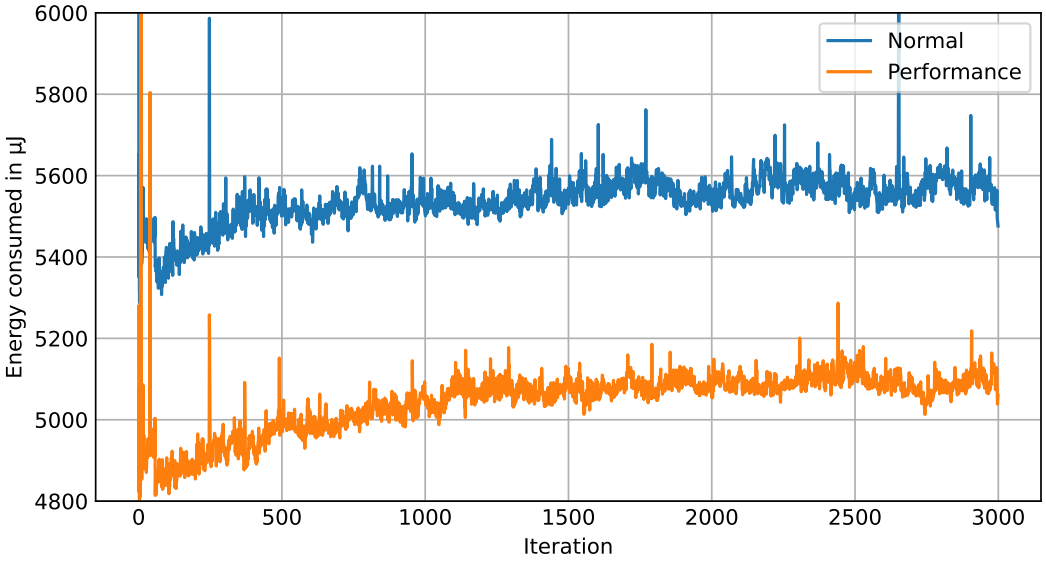

Figure 4.3: Average energy consumed for Newtonsoft.JSON test on the package domain, with total number of runs $m_{performance} = 2\,265\,471, m_{normal} = 2\,044\,789$, p-value = 0.0

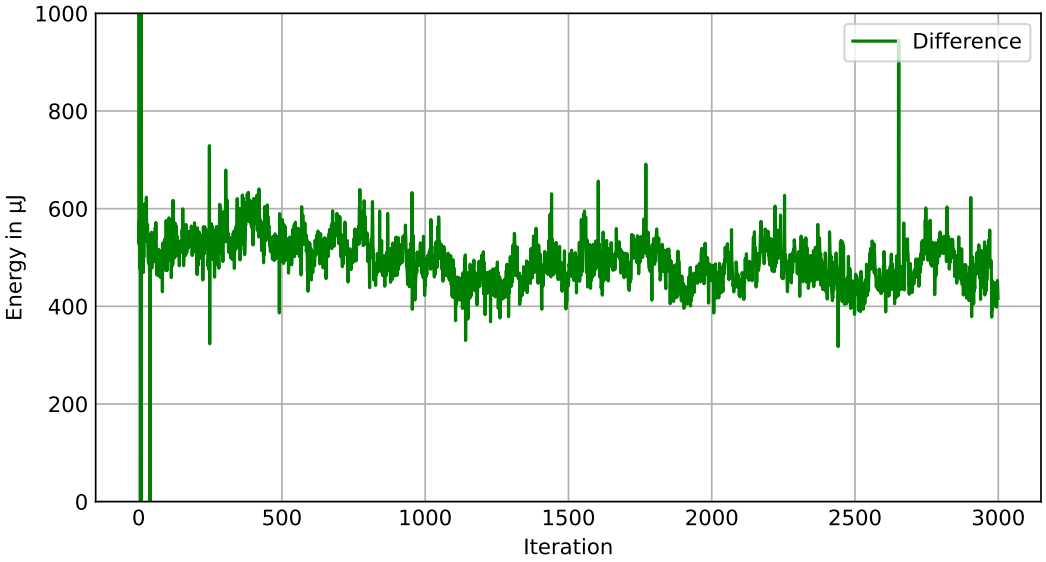

Figure 4.4: Difference in energy consumed for the package domain

The package domain follows the core domain, but with an offset. This was expected as the program primarily uses the CPU, which is what the core domain measures and the core domain is included in the package domain.

4.3.2 Average time per iteration

As mentioned in section 3.2 the execution time is also measured. The average run time for each iteration is plotted in Figure 4.5. Additionally, the difference in run time per iteration is shown in Figure 4.6.

The interesting results from measuring the time elapsed, is that it is much more consistent, compared the energy consumption. The first $\approx$ 100 results are a bit higher than the remaining. The remaining results have very little spread. Additionally when comparing the energy graph and the time graph, the energy graph did have a rising energy consumption for the first $\approx$ 1000 iterations. The time graph does not have this.

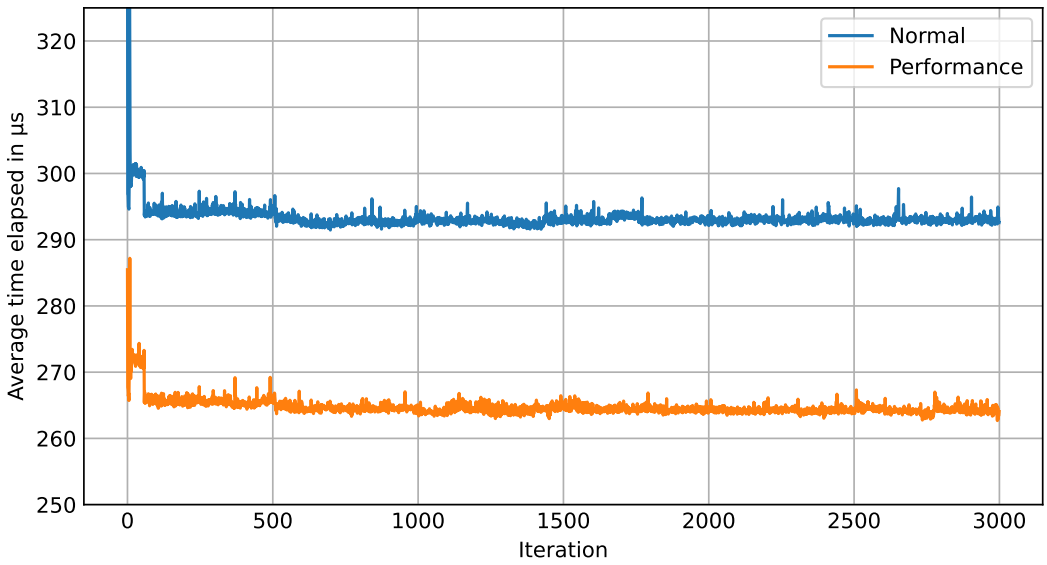

Figure 4.5: Average time over iterations for Newtonsoft.JSON test, with total number of runs $m_{performance} = 2\,265\,471$, $m_{normal} = 2\,044\,789$, p-value = 0.0

Metric	**Normal (mean)**	**Performance (mean)**	**Diff.**	**Percent diff.**
Time	294.16 μs	261.80 μs	32.36 μs	11.00 %
Core	5 075.75 μJ	4 627.14 μJ	448.61 μJ	8.83 %
Package	5 540.69 μJ	5 046.54 μJ	494.15 μJ	8.92 %

Table 4.2: Differences between normal mean and performance mean

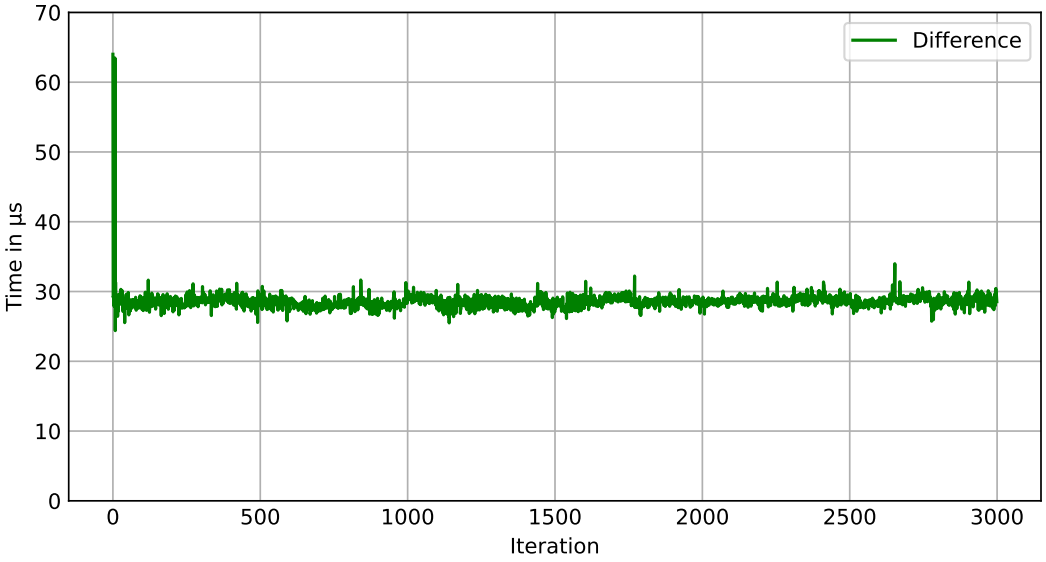

Figure 4.6: Difference in time

The differences between the means of normal and performance commits are shown in Table 4.2. Here it is seen that there is a larger difference in time compared to energy usage, and performance improvements are therefor not one-to-one.

4.3.3 Energy consumption distributions

In addition to the energy consumption over time, it is also interesting to look at a histogram of the data. As seen in Figure 4.7, there is a distinct peak for each of the tested versions. Both results are also negatively skewed towards a lower energy consumption, however from the previous Figure 4.3 we know that these happened earlier in the test.

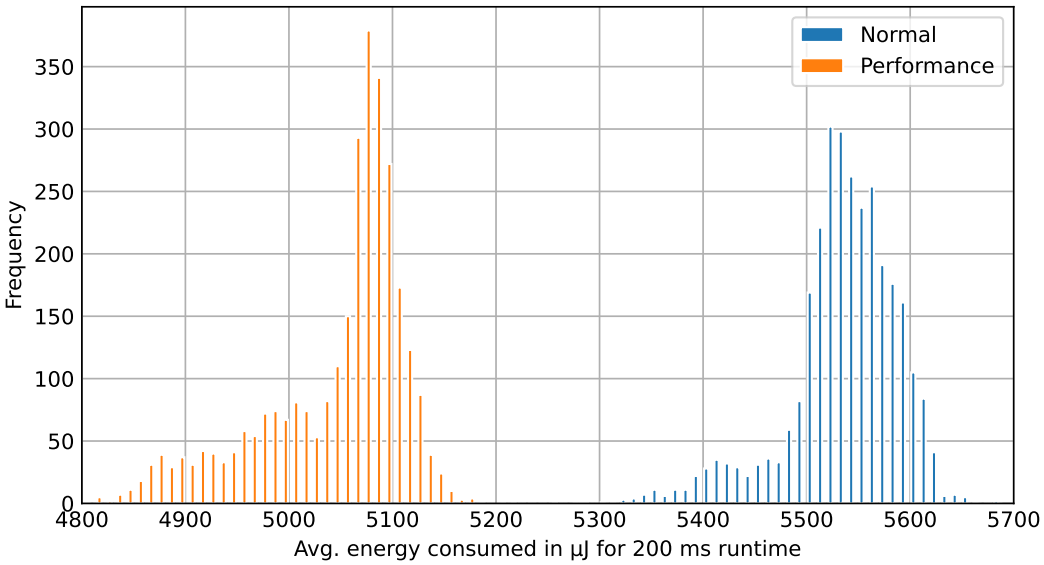

Figure 4.7: Energy consumption for Newtonsoft.JSON test for the package domain, with total number of runs $m_{performance} = 2\,265\,471, m_{normal} = 2\,044\,789$, p-value = 0.0

4.4 Summary

Through implementing a test for serializing integers, we are able to detect a change in energy consumption and in run time. The change is $\approx 30\mu s$ in a test which is $300\mu s$ in run time. There is an increase in the energy consumption over the first 1000 iterations, while this phenomenon is not present in the run time. The test shows an 11% reduction in run time and a $\approx$ 9% reduction in energy usage.

Chapter 5

Newtonsoft: Fast reverse

In chapter 4 the method for doing energy measurements with unit tests was reviewed. We found interesting results which did show a difference in both energy and time consumption. In order to ensure that this is not a coincidence, we conduct a second experiment on a different case.

5.1 Setup

The basis for this experiment is to apply the same approach as in chapter 4, find a performance commit, test the commit and comment on the results. The line in the changelog which this chapter regards is *New feature - Improved performance of JToken.Path with a faster reverse*. The changelog even includes a reference to an issue which was raised by a Microsoft employee on Github[1]. This issue includes argumentation for including a function *FastReverse* which according to the reporter is a substantial improvement for .NET Framework.

A little background knowledge of .NET is required to understand this. Microsoft had .NET Framework which is Microsoft's software framework for writing C#. .NET framework only works on Windows. Later Microsoft released .NET Core which is an open source, cross platform software framework for writing C#. In 2020, Microsoft announced[2] that the .NET versions (Framework and Core) would be unified under the same name, .NET and that it would be a rebranding of .NET Core. This issue only relates to how Newtonsoft.JSON works in regards to the .NET Framework since .NET Core implemented a fix to solve this problem already. It should be noted that Newtonsoft.JSON is released in many versions, all the way from .NET Framework 2.0 to the newest .NET version, including the new .NET Core versions. Therefore, the author of the Github issue suggests this fix

[1] `https://github.com/JamesNK/Newtonsoft.Json/issues/1430`

[2] `https://devblogs.microsoft.com/dotnet/announcing-net-5-0/`

for Newtonsoft.JSON's .NET Framework build. James Newton-King, the author of Newtonsoft.JSON, therefore includes the fix for .NET Framework by utilising an environmental variable and C# preprocessor directives to implement the functionality for the different builds.

An important note regarding the fix is that both the issue and the commit were made in September, 2017. At that time, .NET Core 2.0[3] and .NET Framework 4.7[4] were the newest versions. Since the tests are to be conducted on linux, .NET Framework is out of the question. Instead we want to see if we can detect any difference in energy consumption on .NET Core. We expect to see no difference since the fix for .NET Core[5] is implemented. We build the solution with the commit before the `FastReverse`-functionality and the commit which adds the functionality, with the environmental variable present.

5.1.1 FastReverse

Before doing any testing, we need to establish what the `FastReverse`-function does. Newtonsoft.JSON has a function which returns the path of a JSON object in regards to the whole JSON object. For an object nested twice within an array, the function may return `[0][0]`. In order to compute this, the function appends the current scope to a list and changes the scope to be that of the current scopes parent. The list is then reversed to print in the correct order with regards to the object. It is this reversal of a list which has different implementations. Given that the number of elements which has to be reversed is dependant on the depth of the object, multiple tests can be created in order to see how the implementation is impacted, depending on input. For this purpose, we test the `Path`-functionality with the depths, 1, 100 and 10 000 nested arrays.

5.1.2 FastReverse commit results

The results from this `FastReverse` test can be seen in Figure 5.1, Figure 5.2 and Figure 5.3 for core, package and time respectively. These shown results are all for the 10 000 nested arrays and the 1 and 100 test results can be seen in Appendix D. The graphs are very focused on the distribution, as can be seen on the ranges on the x-axis.

While the p-values show that the results are statistically significant, the sample size is 3000 which could mean that the p-value problem is occurring. The results are also very clustered and not as clear cut as the previous result. Additionally, the up to 50% increase which was described in the Github issue can not be seen. This

[3] `https://devblogs.microsoft.com/dotnet/announcing-net-core-2-0/`

[4] `https://devblogs.microsoft.com/dotnet/announcing-the-net-framework-4-7-general-availability/`

[5] `https://github.com/dotnet/coreclr/pull/1231`

was as expected, but the fact that the `FastReverse` uses more energy and time than the normal reverse is surprising.

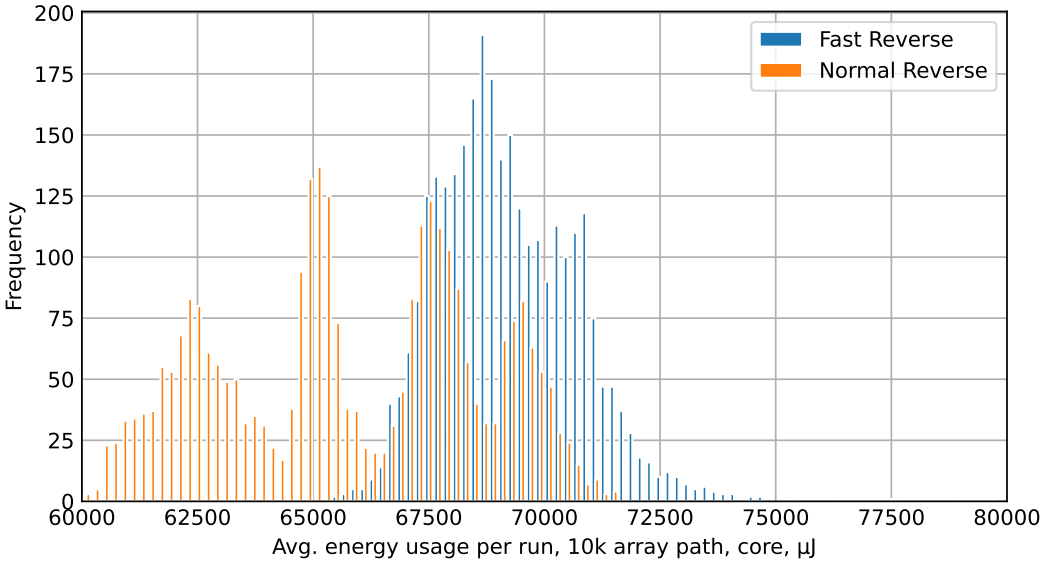

Figure 5.1: Core domain for path depth 10 000, with total number of runs $m_{fast} = 143\,291, m_{normal} = 144\,985$, p-value = 0.0

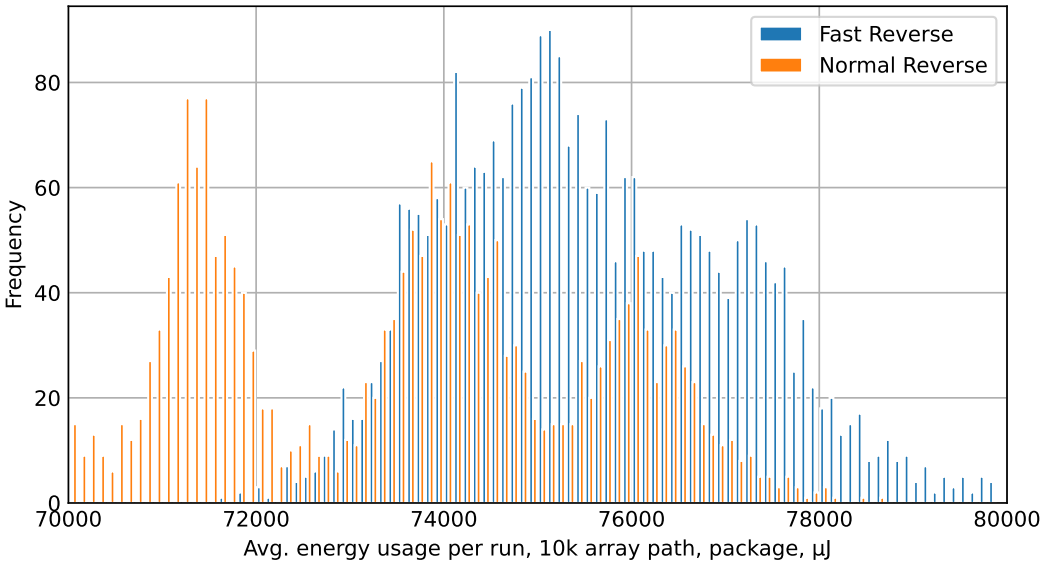

Figure 5.2: Package domain for path depth 10 000, with total number of runs $m_{fast} = 143\,291, m_{normal} = 144\,985$, p-value = 0.0

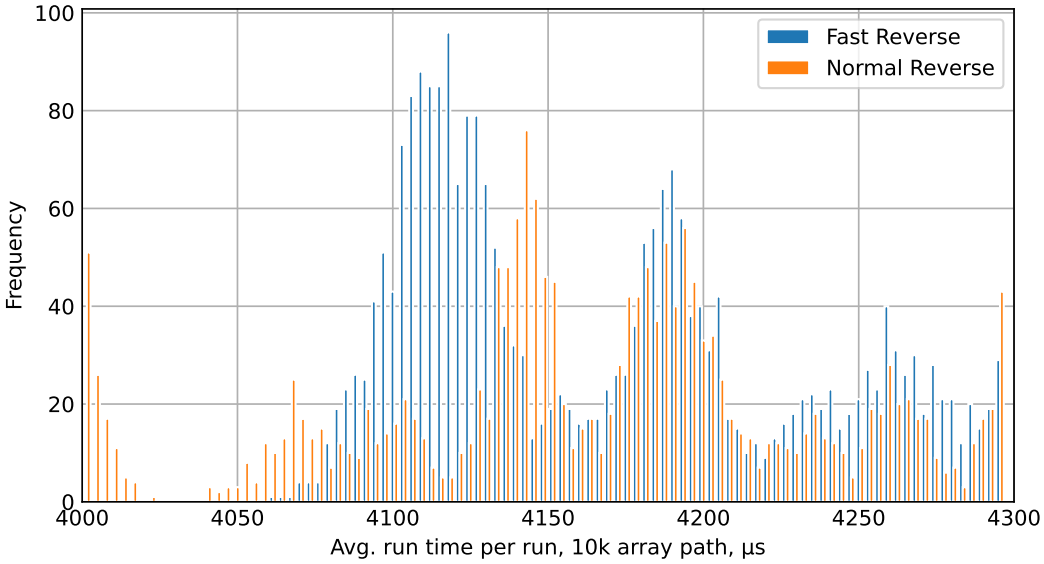

Figure 5.3: Time for path depth 10 000, with total number of runs $m_{fast} = 143\,291, m_{normal} = 144\,985$, p-value = 1.61e-32

Metric	**FastRev (mean)**	**NormalRev (mean)**	**Diff.**	**Percent diff.**
Time	4 189.05 μs	4 141.10 μs	47.95 μs	1.14 %
Core	69 218.59 μJ	65 822.31 μJ	3 396.28 μJ	4.91 %
Package	75 594.56 μJ	72 130.25 μJ	3 464.31 μJ	4.58 %

Table 5.1: Means and their differences for the test case of 10k nested array

5.1.3 .NET 6

Upon researching the implementation and how the library is built today, .NET 6.0 was discovered to include the flag for compiling with `FastReverse` which is unexpected since .NET Core already had implemented the fix mentioned in the issue. In the commit[6], there is no mention on why this was implemented that way, but the commit has ≈ 900 changes. The flag in question is the `HAVE_FAST_REVERSE`. However when talking about a single flag, it is also important to note that in total, Newtonsoft.JSON for .NET 6.0 has 61 different build flags with support to adding a custom number of additional flags.

This raises the question:

Is it possible to detect a change between Newtonsoft.JSON in .NET 6 built with and without the flag `HAVE_FAST_REVERSE`?

[6] `https://github.com/JamesNK/Newtonsoft.Json/commit/bf2e2a78e8febf0006ec647f9bde3aa5bbe0ce72`

The process is to build Newtonsoft.JSON with/without the flag and then running the tests of fetching the path of an object in question. The tests from subsection 5.1.1 with the sample sizes of 1, 100 and 10 000 nested arrays are re-utilized in order to test the library.

Between the implementation of `FastReverse` and the .NET 6 target, Newtonsoft.JSON has implemented a default value for the `MaxDepth`-property. This property controls how deep an object the library is able to parse before throwing an exception. The reasoning for implementing such a default is that there could be a potential Denial of Service attack[7]. In order to run our tests with depth 100 and 10 000 the default needs to be overwritten. This is done by constructing the JSON deserializer with the `MaxDepth` option set to `NULL`.

5.2 Results

In this section the results of the .NET 6 tests outlined in subsection 5.1.3 are presented and discussed. The results are divided into three subsections, for each of the test depths.

5.2.1 Test case with depth 1

For the 1 depth test case the results for core, package and time can be seen on figures Figure 5.4, Figure 5.5 and Figure 5.6 respectively. Here it can be seen that there is a distinct difference between the two test results, albeit a small one. The total percentage difference can be seen in Table 5.2 and it varies between the time and energy domains. In all three cases the distributions are normal and for the two energy distributions there is a slight left (negative) skewness.

[7] `https://github.com/JamesNK/Newtonsoft.Json/pull/2462`

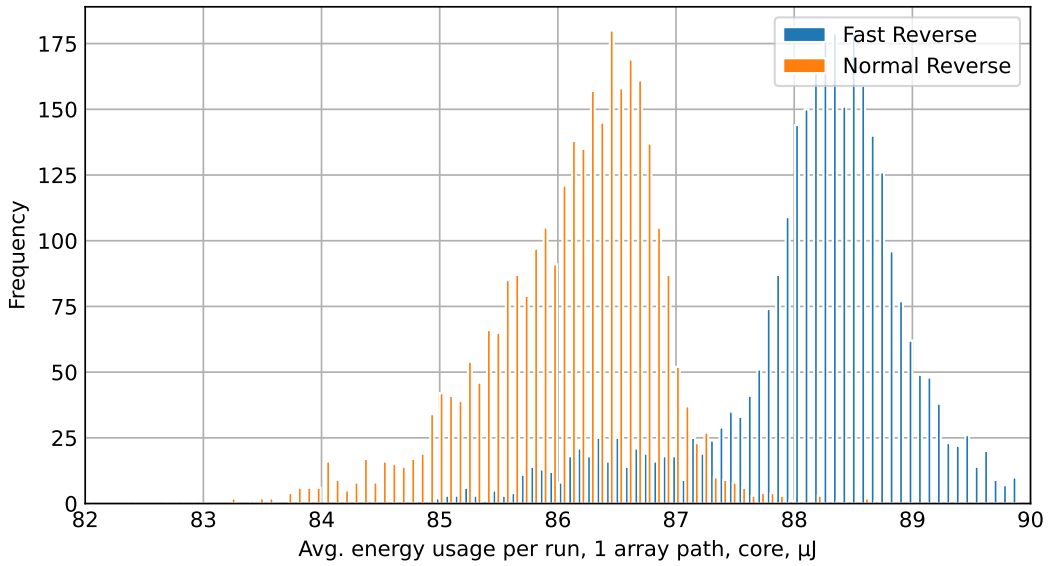

Figure 5.4: Core domain for path depth 1, with total number of runs $m_{fast} = 113\,764\,904, m_{normal} = 118\,497\,335$, p-value = 0.0

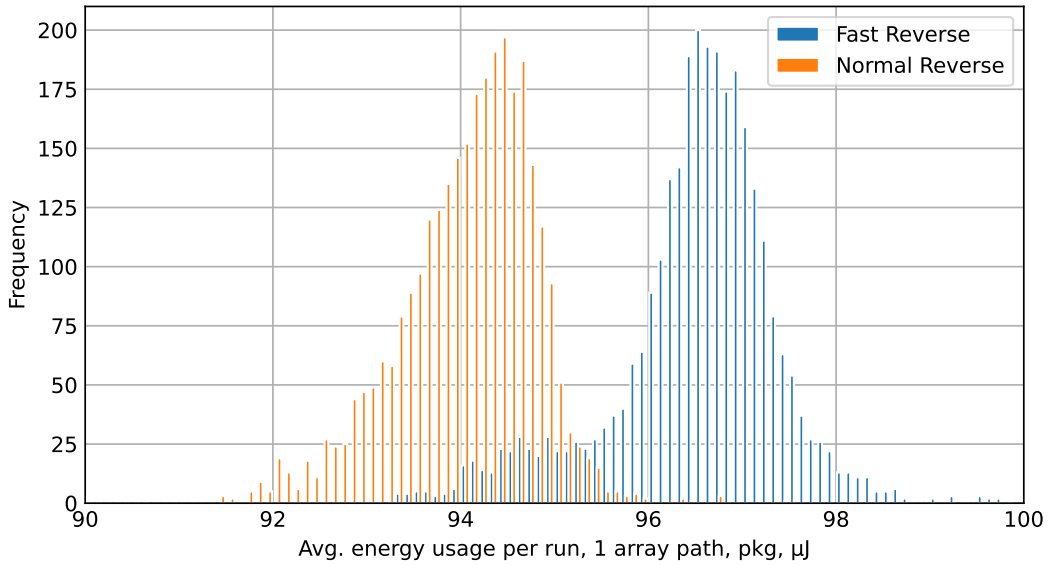

Figure 5.5: Package domain for path depth 1, with total number of runs $m_{fast} = 113\,764\,904, m_{normal} = 118\,497\,335$, p-value = 0.0

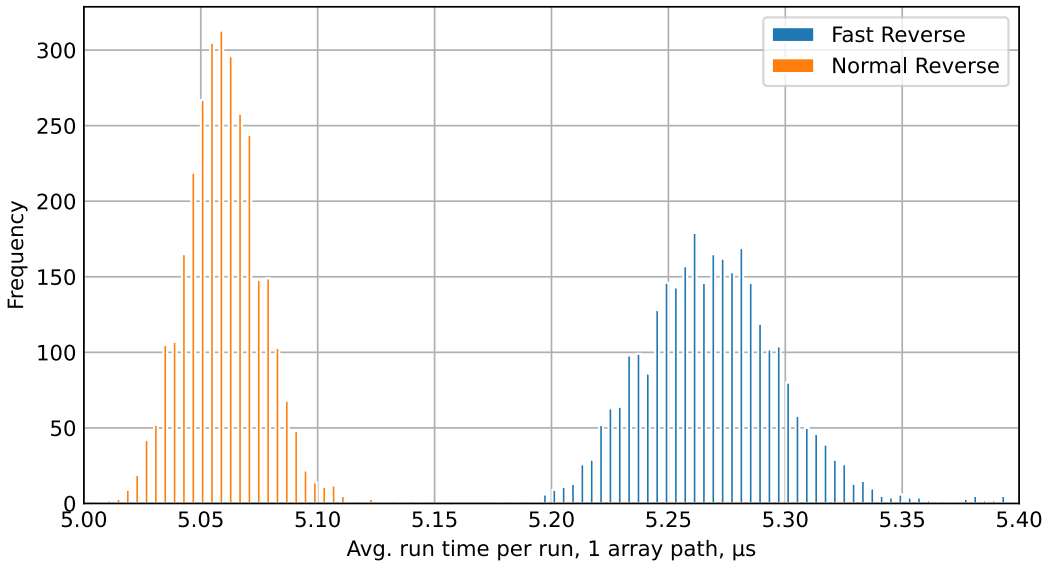

Figure 5.6: Time for path depth 1, with total number of runs $m_{fast} = 113\,764\,904, m_{normal} = 118\,497\,335$, p-value = 0.0

Metric	**FastRev (mean)**	**NormalRev (mean)**	**Diff.**	**Percent diff.**
Time	5.27 μs	5.06 μs	0.21 μs	3.98 %
Core	88.28 μJ	86.18 μJ	2.10 μJ	2.37 %
Package	96.59 μJ	94.14μJ	2.45 μJ	2.53 %

Table 5.2: Means and their differences for the test case of an array

5.2.2 Test case with depth 100

For the 100 depth test case the results for core, package and time can be seen on figures Figure 5.7, Figure 5.8 and Figure 5.9 respectively. The total percentage difference can be seen in Table 5.3 and it varies between the time and energy domains, but to a lesser degree than for the depth 1. In all three cases the distributions are normal as well.

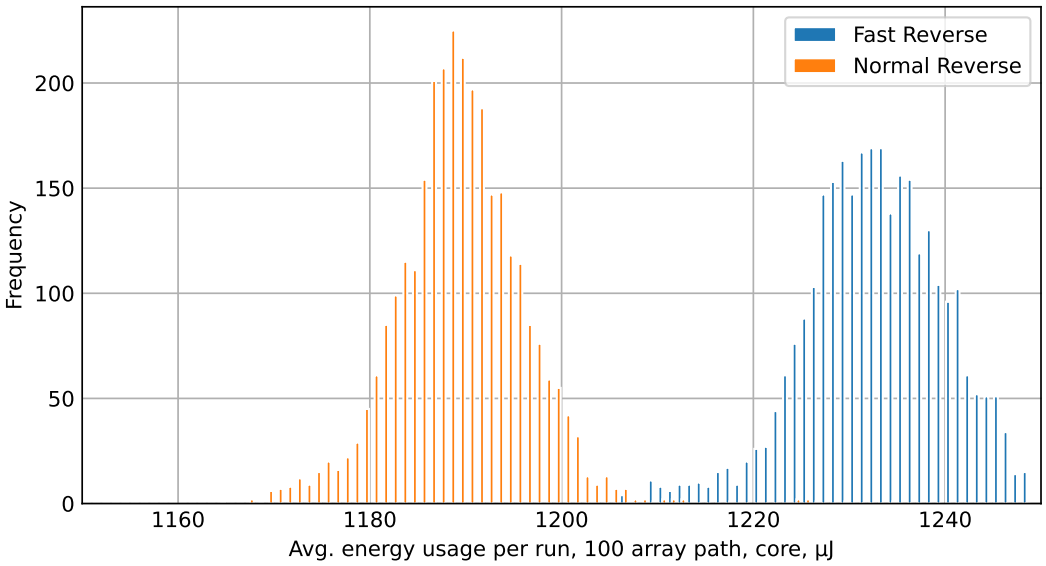

Figure 5.7: Core domain for path depth 100, with total number of runs $m_{fast} = 9\,312\,954, m_{normal} = 9\,615\,599$, p-value = 0.0

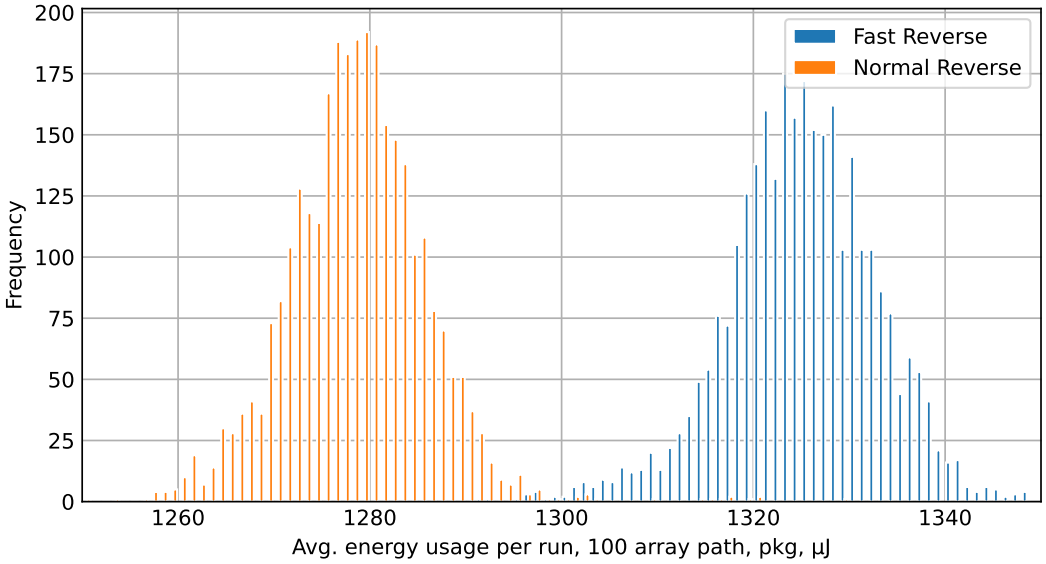

Figure 5.8: Package domain for path depth 100, with total number of runs $m_{fast} = 9\,312\,954, m_{normal} = 9\,615\,599$, p-value = 0.0

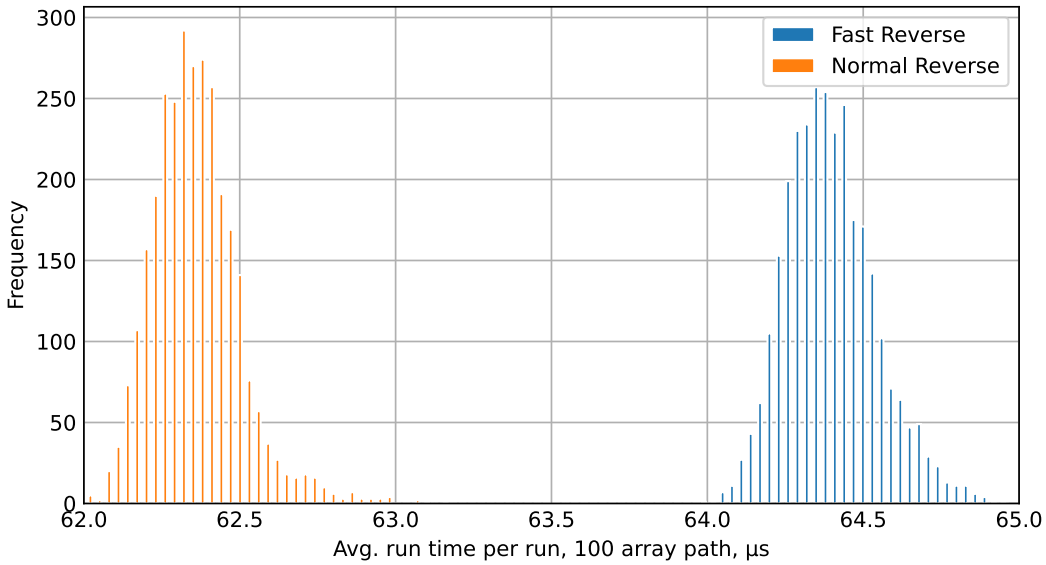

Figure 5.9: Time for path depth 100, with total number of runs $m_{fast} = 9\,312\,954$, $m_{normal} = 9\,615\,599$, p-value = 0.0

Metric	**FastRev (mean)**	**NormalRev (mean)**	**Diff.**	**Percent diff.**
Time	64.41 μs	62.36 μs	2.05 μs	3.18 %
Core	1 233.35 μJ	1 189.63 μJ	43.72 μJ	3.54 %
Package	1 325.49 μJ	1 278.77 μJ	46.72 μJ	3.52 %

Table 5.3: Means and their differences for the test case of 100 nested array

5.2.3 Test case with depth 10 000

For the 10 000 depth test case the results for core, package and time can be seen on figures Figure 5.10, Figure 5.11 and Figure 5.12 respectively. The total percentage difference can be seen in Table 5.4, however the difference here is smaller than both of the previous cases which is surprising as the expectation was that scaling the experiment would amplify the differences. Additionally, the energy distributions are normal, but the run time distribution is multi modal for both reverse functions, with a larger secondary peak for the fast reverse than for the normal reverse.

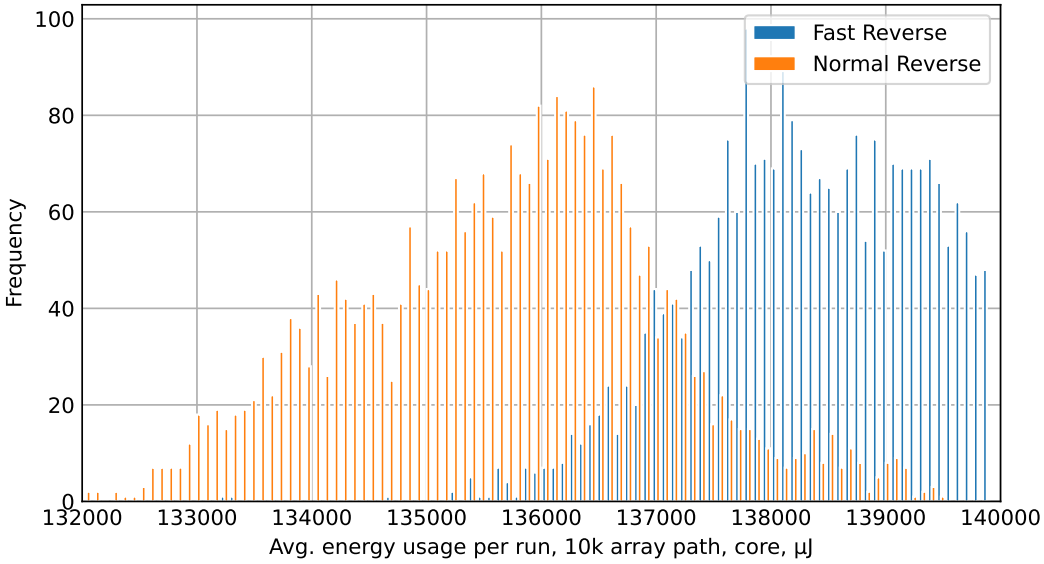

Figure 5.10: Core domain for path depth 10 000, with total number of runs $m_{fast} = 80\,356, m_{normal} = 81\,969$, p-value = 0.0

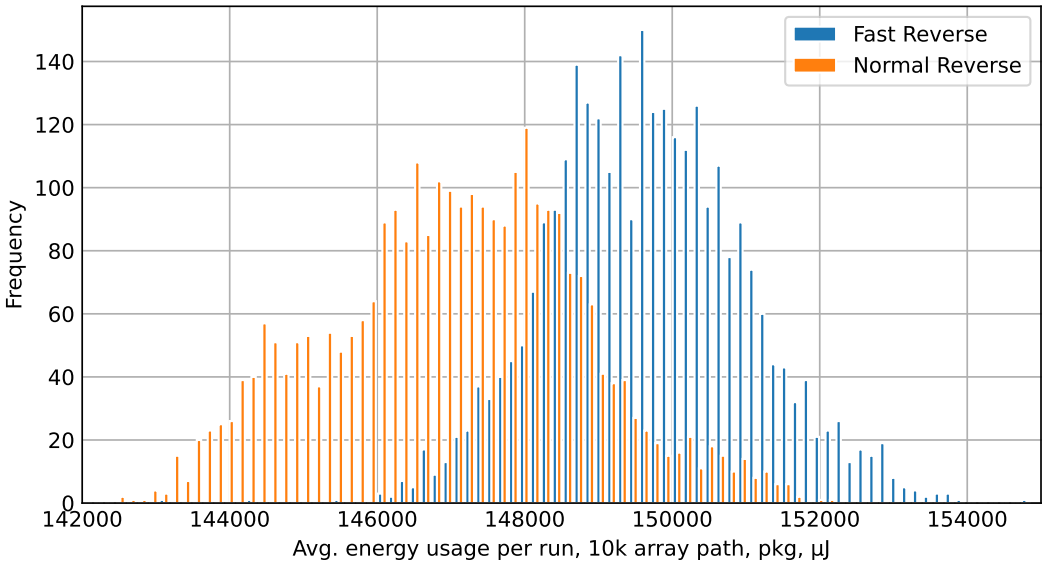

Figure 5.11: Package domain for path depth 10 000, with total number of runs $m_{fast} = 80\,356, m_{normal} = 81\,969$, p-value = 0.0

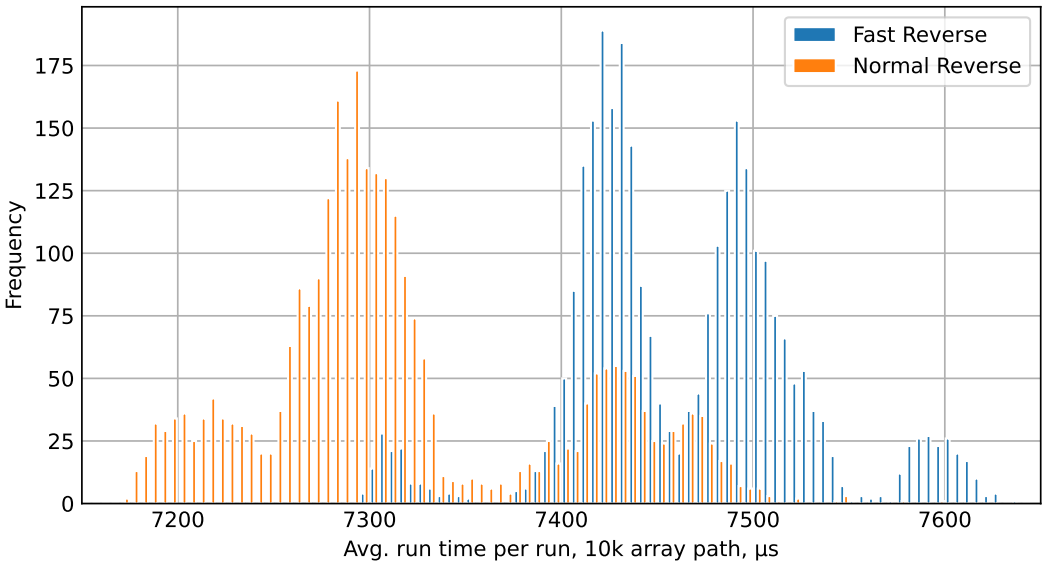

Figure 5.12: Time for path depth 10 000, with total number of runs $m_{fast} = 80\,356, m_{normal} = 81\,969$, p-value = 0.0

Metric	**FastRev (mean)**	**NormalRev (mean)**	**Diff.**	**Percent diff.**
Time	7 464.04 μs	7 318.59 μs	145.45 μs	1.95 %
Core	138 614.99 μJ	135 707.84 μJ	2 907.15 μJ	2.10 %
Package	149 681.32 μJ	147 049.77 μJ	2 631.55 μJ	1.76 %

Table 5.4: Means and their differences for the test case of 10k nested array

An interesting observation is that compared to the other version where commits were compared, the time and energy consumption is significantly higher. The numbers in question are presented in Table 5.1 and in Table 5.4. For the .NET 6 version for a nested array of 10 000 depth each iteration required on average $\approx 7.5ms$, for the earlier version, the average iteration required $\approx 4.2ms$. This is a difference of over 75 %. For the energy consumption there is also a major difference. In .NET 6 the average energy consumption for the core domain is $\approx 139\mu J$ while in the earlier version, the average energy consumption is $\approx 69\mu J$. The reason for this is unknown and needs further investigation. However the change could be due to architectural changes or due to the fact that the tests are not entirely conducted in the same way, as mentioned in subsection 5.1.3.

The important thing to note from these results is not only the size of the difference, it is in which direction the difference lies. This is because the normal .NET 6 reverse function is faster and more energy efficient than the custom implemented fast reverse. This means that the legacy code which is kept by using the same

compiler flags for newer versions of the Newtonsoft.JSON library is negatively affecting the performance of the library.

5.3 Summary

We were able to detect a difference in energy consumption when comparing before and after the commit in subsection 5.1.1. The energy consumption was 5 % higher, while the run time was 1 % higher in the `FastReverse`-function compared to the Normal Reverse. However these results were very close to each other and deemed not significant.

When comparing the inclusion of the build flag in .NET 6, we did see a difference in energy consumption.

For a depth of 1, the change in energy consumption of the core domain was $\approx$ 2%, the package domain was $\approx$ 3% and the change in run time was $\approx$ 4%. Depth 100 had a change in energy consumption of $\approx$ 4% for the core domain, the package domain was $\approx$ 4% and the change in run time was $\approx$ 3%. For the final depth of 10 000 the energy consumption for the core domain had a change of $\approx$ 2%, the package domain of $\approx$ 2% and the change in run time of $\approx$ 2%.

Chapter 6

Evaluation

6.1 Discussion

In this section the methodology of the experiments as well as the results and their significance are discussed.

6.1.1 Choosing time over number of iterations

In subsection 3.13.1 it was decided that measurements should be conducted with a time limit instead of a number of iterations. The runs could be conducted by performing a small sample of iterations and from there calculating how many runs should be conducted within a certain time frame. The idea behind using time is to be able to integrate the method into a continuous integration framework. The current solution does not reflect a real world implementation in the aspect that the tests need to be individually configured to last the set amount of time. This process could be automated in order to make the tool more accessible, by providing a set amount of time available for testing. This time would then be divided between all the tests evenly.

In our tests, no obvious problems with using time based testing were encountered. However, this method of conducting performance tests is not common in the related literature. It was also possible to find differences through this methodology, which means that performance improvements can be detected when doing time based testing.

6.1.2 Averaging energy consumption over time

In chapter 3 we found that if the Intel RAPL sampling rate is not accounted for, we can get inaccurate readings of Intel RAPL. As discussed in subsection 4.2.3 we found that the answer is to average the readings over multiple runs. An interesting aspect is that other energy benchmarking frameworks like [20] and [17] do not

account for these limitations. However, this method of averaging is not without its own negatives. The problem here is if there are major slowdowns or speedups for a single iteration, the granularity of that single iteration is lost and this slowdown is divided over the entire average.

6.1.3 Time vs energy optimizations

In section 4.3 the results showed improvements in both run time and energy consumption. The optimization for the library was made with only time improvement in mind, however the energy consumption also improved, as shown by our tests. This is expected as often the two factors are related. The results also showed that while there was an improvement to both factors, they were not one-to-one improvements, as also previously described. The specifics where an 11% time improvement to a 9% energy improvement, based on mean values.

6.1.4 Fibonacchi testing

First it was found that we could measure the energy usage of tests through the Fibonacci experiment in section 3.7 but that the run time was too small in some cases to get an energy reading. This was evident by the use of two different Fibonacci functions, with different complexities, which showed that slower function could be measured while the faster one returned an energy reading of zero, which meant the RAPL counter did not have time to update in the shorter runtime.

6.1.5 Limits of RAPL

This lead us to testing the limitations of the RAPL energy reading framework, and specifically the limitations of our test environment. Here we found a large difference between measuring the core domain and the package domain, where core refers only to CPU energy usage and package the total energy usage which includes CPU, uncore and DRAM. The difference being an update rate of about 1ms for the package domain and a $60\mu s$ update rate for the core domain. Additionally, we tested if this core update rate was accurate, and not just the speed of loop in C#. This was done to verify the previous results for the sample rate.

These results were used in section 3.12 to calculate how small tests could be performed, in regards to testing time, while still achieving satisfying results. However, this was not tested further, but is discussed as future work.

6.1.6 Newtonsoft: Integer serializer

The next experiment which was performed was the integer serializer test in the Newtonsoft.JSON library. Here two commits were tested, one just prior to a per-

formance commit and the other the performance commit itself. The results showed an improvement in both time, which was expected as it was stated as such in the commit but also a energy performance improvement. As previously discussed these improvements were 11% time improvement and 9% energy usage improvement. These results were very clear, which was also evident from Figure 4.7 which shows the two distributions clearly of-set from each other.

6.1.7 Newtonsoft: Reverse, commit version

Following the first Newtonsoft.JSON library test, a second test was performed in an attempt to achieve similar results. However, this test presented some problems with using the older versions of .NET. This was due to the found performance commit being a specific issue to the windows .NET framework. We were unable to test this specific improvement, as the test environment we utilize needs to run Linux in order to measure the RAPL counters. Instead we tested the commit versions for .NET Core since the Github issue for the performance improvement stated that this .NET version already had the optimized array reverse function integrated. As expected we did not find much difference, except that the native reverse was both slightly faster and more energy efficient.

6.1.8 Newtonsoft: Reverse, .NET 6

The results which showed that the fast reverse functionality was slower than the native implementation caused us to investigate this further, as we discovered that the fast reverse compile flag was still present for newer .NET versions, namely .NET 6. To investigate this we made two similar builds for the live version of the Newtonsoft.JSON library with only one difference, rather the compile flag for fast reverse was included or not. One other difference had to be made to the build as well, which could impact the results. This was to overwrite the maxDepth for the nested arrays, which was changed between the commit versions we tested in the previous test and the live version of the library. However, this overwrite had to be done for both .NET 6 versions and it was made to allow us to run our custom tests. This test showed that the fast reverse functionality was both a time and energy performance loss compared to relying on the default .NET 6 reverse functionality.

6.1.9 P-values

Throughout the report the p-values have been presented with the respective graphs. The meaning of p-values was originally presented in section 2.14 and it has not been commented on further since in every case the p-value has been 0.0 or very close to it, which is well under the 0.05 threshold. In chapter 3 we expected the p-values to be higher, since the graphs and measurements where almost identical,

and while the p-values were above 0.0 here, they were still incredibly small. This is most likely due to the p-value problem, which was described in section 2.15, as the experiments have sample sizes in the millions.

6.1.10 Continuous Integration tool

The goal was originally to create this energy testing framework and integrate it into a Continuous Integration workflow, to allow easier adaptation of energy testing for regular developers. While the testing has been successful, the goal of adding it to a Continuous Integration pipeline was not met. The reason this goal was not met is in part due to time spent investigating the specific properties and limitations of Intel RAPL. These abilities and limitations needs to be pushed further than they previously have been, if a Continuous Integration energy tool is to be developed. We still believe this goal is realistic and can be achieved, but previously mentioned considerations regarding testing time and energy test suite sizes will need to further tested and fine tuned to make this goal a reality.

6.2 Threats to validity

In this section potential threats to the validity of the results are presented and the impact and significance of these threats are discussed.

6.2.1 Setup and tear down of test environment

During the initial Fibonacci test, an increased runtime of the unit test was found, compared to the production runtime. This was an issue when running the code using rapl.rs which as mentioned is a wrapper for the energy measurements. This was mitigated later on by adding the energy measurements into the testing framework.

6.2.2 Power domain

Throughout the report there has been different experiments for the core and package domains. It was found that the sample rate of the core domain is more frequent than the package, and it was therefor used to accrue a more accurate reading in regards to update of the RAPL measurement. This does however mean that there can be changes in DRAM or uncore energy usage which goes unmeasured. The importance of this difference is heavily minimised when using averages of a 200ms run time.

6.2.3 Overflow of RAPL counter

In our previous work, a lot of effort went into ensuring that overflow of RAPL counters were handled. The current implemented solution does not take potential overflows into account. The reason being that if an overflow was to happen, the average would be negative, and cause a distinct outlier on the graphs. However further investigation is required into how an overflow would look in order to perform the necessary actions. Having an overflow check in the code would add additional overhead, which is undesirable when working with tests as small as these.

6.2.4 Override MaxDepth on FastReverse

As mentioned in subsection 5.1.3, Newtonsoft.JSON had implemented a fix for a potential Denial of Service attack by introducing a default *MaxDepth*-value. This value was overwritten in order to perform the same tests on the .NET 6 version as was conducted on the older versions.

6.2.5 Large sample size with Mann-Whitney U test

In our experiments we have attempted to verify the statistical significance though the use of the Mann-Whitney U test. In every experiment the p-value has been far below the probability threshold. In most cases this was to be expected as there were large differences in the measured values. However, it is uncertain if these p-value results are to be trusted, since we are working with large sample sizes which causes the p-value problem, which was presented in section 2.15, and previously discussed in subsection 6.1.9.

Chapter 7

Conclusion

The aim of this project was to help programmers become more aware of the energy usage of their code. The motivation for this was the company Edora which wanted to monitor their energy consumption. The project focused on C# since this area is lacking when it comes to energy consumption research.

This motivated the development of an energy measuring framework built into MSTestV2 such that the energy measuring of a system could be automated in the same way regular testing is in build frameworks. This was with the end goal of integrating energy testing into a Continuous Integration framework and automating the process of creating energy distributions and differences in energy usage of a system, in a similar fashion to how time based performance testing is done today.

We set up an experiment using a small Fibonacci program in order to test the initial version of the MSTestV2 framework. Upon examining the results, it was found that the readings often read 0 energy consumption. From there we experimented with the sample rate of Intel RAPL in order to determine the update frequency, which was found to be $60\mu s$ for the core domain and $1080\mu s$ for the package domain. Additionally, an experiment was performed to verify that C# captures every individual RAPL update event, and it was discovered that C#s loop rate is about five times faster than the RAPL update rate. In the initial experiments, we experimented with Intel SGX being enabled or disabled and found no differences in the RAPL update frequency.

Following these experiments, a larger system was tested, namely the Newtonsoft.JSON library. Here performance commits were identified through the change log and unit tests were run on the functionality before and after the given change to specify the significance and quantity of the performance improvement. For the first Newtonsoft experiment the difference between the two versions was 11% improvement in run time and a 9% improvement in energy usage.

To expand on this success, we selected another change log entry to examine

its energy impact. We selected a functionality which is related to how an array is reversed. While this commit was aimed to make an improvement on the Windows version, we decided to see if any difference could be identified on the Linux version, since the testing framework using RAPL only works in Linux. In conducting the experiment, we found an increase of 1 % in run time and 5 % increased energy consumption, however not as clear a distinction as the previous experiment.

On researching the newer versions of the library, it was discovered that in the newest .NET 6, the build flag was still present despite the original issue being related to .NET Framework. This was surprising and we experimented with having the build flag present. In our findings, the build flag made a 2-4 % run time change and a change of 2-4 % in energy consumption, depending on the depth of the test. Thus we conclude that the performance was worse when the build flag was present.

The goal of creating a Continuous Integration solution was not achieved, but part of the foundation required for such a solution has been laid out in this report. This basis allows for an overnight energy testing framework to be created. A Continuous Integration tool would require additional research and experimentation into minimizing the run times and iterations of energy tests, while still achieving sufficient and accurate results.

Chapter 8

Future work

In this chapter the possibilities for future work using the work presented in this report is discussed.

8.1 Examine the reason for a rise in energy consumption despite constant time consumption

In subsection 4.3.1, the energy was increasing over the first $\approx$ 1000 results. This is despite the time consumption being almost constant. Could this be due to the hardware getting hotter, increasing the resistance and thus requiring more energy to keep the same level of performance? A possibility for testing this could be to record each core temperature and review the correlation to test this hypothesis. It could also be tested by adding a cool down period in the middle of the test run time. However, if this is added to production this would mean having less time to perform tests when aiming to integrate it into the previously talked about 10 minutes of run time.

8.2 Integrating the testing measurements into Continuous Integration or BenchmarkDotNet framework

The current solution investigates how to perform measurements with Intel RAPL. For appealing to a broader audience, the methods presented here could be implemented into a framework like BenchmarkDotNet[1]. The idea being that it would lower the barrier of entry and enable developers to benchmark their code with regards to energy consumption.

[1] `https://benchmarkdotnet.org/articles/overview.html`

8.3 Investigate project size cut off point between Continuous Integration and overnight testing

Tests are limited to 10 minutes in this report. However larger projects may have many hundreds of tests which could mean that in order to test them all, the timeframe of doing energy measurements may need to be extended. This timeframe could be an overnight testing or maybe in the weekend where there may be fewer developers working on the project.

8.4 Develop a windows version using Intel Power Gadget

A limitation for the current version is that it depends on the powercap integration to Intel RAPL. This integration is only present on linux based systems. Intel does provide a *Power Gadget*-tool which can record energy consumption on Windows machines. An investigative work on Intel Power Gadget could explore the possibilities and limitations of the application and maybe end up with a solution which can perform energy benchmarks on Windows.

8.5 Replicate results on different test environments

The work in this report is only using a single test environment. Intel RAPL is available on most systems since 2012 and AMD is now also implementing their version. Further work should go into exploring the test results on different test environments. This would be to explore if there are differences in the RAPL implementation or maybe even differences in the energy consumption for the tests conducted.

8.6 Investigate JIT, compiler optimization and garbage collection impact on results

In section 4.3, an early spike can be seen. This could possibly be due to the Just-in-time compilation of .NET 6, where the first couple of measurements are higher than average. An example experiment could be with regards to the garbage collection which can be controlled through the code. Here the garbage collector could be disabled entirely and see if spikes are less frequent.

8.7 Compare the build flag of the fast reverse commit

As explained in section 5.1, the comparison of energy is between the implemented functionality with the build flag present and the previous commit. For .NET 6, this comparison is between the current version, with and without the build flag present. It could be interesting to test presence of the build flag of the first version. This would make the two experiments more comparable.

8.8 Investigating the minimum test run time

In section 3.7 the Fibbonaci function was tested, here it was discovered that some tests were too fast for the RAPL counter to update in time. This was further investigated in section 3.12, where two minimum values for what the run time of a test should be, for core and package domains respectively. However, these values were not tested themselves. In fact, in subsection 4.2.3 we decide to use a higher value than calculated, in order to be extra safe and avoid any problems with the readings. These found values for minimum test run time should be tested and would then allow for shorter testing time. This would in turn make it easier to have as many performance tests as possible within a 10-minute window. This 10-minute window was the aim in order to integrate the testing into a continuous integration framework as previously presented in section 3.13.

8.9 Examine the accuracy impact of Intel SGX

In section 2.7, Intel's response to the platypus attack was documented. The update frequency of Intel RAPL with SGX status being changed was examined in chapter 3 where we found no difference to the update frequency regardless of the SGX status. According to Intel the updated micro code also includes some accuracy changes for the energy readings which should make Platypus style attacks more difficult. An experiment could be set up to determine the impact of this update.

Bibliography

[1] Andrey Akinshin. *Pro .NET Benchmarking*. Apress, 2019. ISBN: 978-1-4842-4941-3. URL: https://doi.org/10.1007/978-1-4842-4941-3.

[2] Jiahao Chen and Jarrett Revels. "Robust benchmarking in noisy environments". In: *arXiv preprint arXiv:1608.04295* (2016). URL: https://arxiv.org/pdf/1608.04295.pdf.

[3] Marco Couto et al. "Towards a green ranking for programming languages". In: *Proceedings of the 21st Brazilian Symposium on Programming Languages*. 2017, pp. 1–8. URL: https://dl.acm.org/doi/pdf/10.1145/3125374.3125382.

[4] Benjamin Danglot, Jean-Rémy Falleri, and Romain Rouvoy. "Can We Spot Energy Regressions using Developers Tests?" In: *arXiv preprint arXiv:2108.05691* (2021). URL: https://arxiv.org/pdf/2108.05691.pdf.

[5] explorable.com. *Bell Cruve graphic*. URL: https://explorable.com/bell-curve-controversy.

[6] Martin Fowler and Matthew Foemmel. *Continuous integration*. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html.

[7] Chris Geiger, Dennis Przytarski, and Sascha Thullner. "Performance testing in continuous integration environments". In: (2014). URL: https://elib.uni-stuttgart.de/bitstream/11682/3311/1/FACH_0188.pdf.

[8] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3*. 2021. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[9] Intel. *Running Average Power Limit Energy Reporting / CVE-2020-8694 , CVE-2020-8695 / INTEL-SA-00389*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[10] Erik Jagroep et al. "The hunt for the guzzler: Architecture-based energy profiling using stubs". In: *Information and Software Technology* 95 (2018), pp. 165–176. URL: https://www.sciencedirect.com/science/article/pii/S0950584917303841.

[11] rust lang.org. *Rust nightly*. URL: https://doc.rust-lang.org/book/appendix-07-nightly-rust.html.

[12] Mingfeng Lin, Henry Lucas, and Galit Shmueli. "Too Big to Fail: Large Samples and the p-Value Problem". In: *Information Systems Research* 24 (Dec. 2013), pp. 906–917. DOI: 10.1287/isre.2013.0480.

[13] Moritz Lipp et al. "PLATYPUS: software-based power side-channel attacks on x86". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 355–371. URL: https://ieeexplore.ieee.org/iel7/9519381/9519382/09519416.pdf.

[14] Microsoft. *C# Compiler Options that control code generation*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/code-generation.

[15] Microsoft. *Understand build configurations*. URL: https://docs.microsoft.com/en-us/visualstudio/ide/understanding-build-configurations?view=vs-2022.

[16] Daniél Garrido-Y Martinez Nielsen et al. "Energy Benchmarking With Doom". In: (2021).

[17] Jacob Ruberg Nørhave, Casper Susgaard Nielsen, and Anne Benedicte Abildgaard Ejsing. "IDE Extension for Reasoning About Energy Consumption". In: (2021). URL: https://projekter.aau.dk/projekter/files/422795208/Kandidat_Projekt__1_.pdf.

[18] Francisco G de Oliveira Neto et al. "Improving continuous integration with similarity-based test case selection". In: *Proceedings of the 13th International Workshop on Automation of Software Test*. 2018, pp. 39–45. URL: https://dl.acm.org/doi/pdf/10.1145/3194733.3194744.

[19] Ian Sommerville. *Software Engineering*. Global, 10th edition. Pearson, 2016. ISBN: 978-1-292-09613-1.

[20] Jeffrey Thijs Raymakers Katja Schmahl. *Coppers - Github*. URL: https://github.com/ThijsRay/coppers.

[21] Wolfram. *Normal Distribution*. URL: https://mathworld.wolfram.com/NormalDistribution.html.

Appendix A

Implementing Intel RAPL in MSTest V2

Implementation of Intel RAPL in MSTest V2.

```
using UtilityLibraries;

class Program {
  private const string FILE_PATH =
  ↪ "/sys/devices/virtual/powercap/intel-rapl/intel-rapl:0/energy_uj";

  private static Decimal read_rapl_value() {
    string raw_value = System.IO.File.ReadAllText(FILE_PATH);

    return Decimal.Parse(raw_value);
  }

  static void Main(string[] args) {
    Console.WriteLine("Iteration;RAPL-Value;Old-RAPL;New-RAPL;Ticks");
    for(int count = 0; count < 25; count++){
      Decimal before_value = read_rapl_value();
      var watch = System.Diagnostics.Stopwatch.StartNew();

      Console.WriteLine(FibLibrary.SlowGetFib(42));

      Decimal new_value = read_rapl_value();
      watch.Stop();
      Console.WriteLine("{0};{1};{2};{3};{4}", count, new_value-before_value,
      ↪ before_value, new_value, watch.ElapsedTicks);
    }
  }
}
```

Snippet A.0.1: Custom attribute for MSTest V2. In this test, we are able to extend the functionality for recording Intel RAPL values on a iterative based method

Appendix B

Fib-experiment

Fibonacchi experiment results

FibLibrary.GetFib(41)			FibLibrary.GetFib(42)			FibLibrary.GetFib(43)		
Iteration	MicroJoule	Nanoseconds	Iteration	MicroJoule	Nanoseconds	Iteration	MicroJoule	Nanoseconds
0	85 510	4 043 225	0	0		0	0	139 351
1	0	165 079	1	0	26 018	1	0	33 651
2	0	117 057	2	0	36 371	2	0	40 608
3	0	43 386	3	0	36 282	3	0	37 636
4	0	38 468	4	0	35 734	4	0	38 759
5	0	39 515	5	0	35 288	5	0	25 753
6	0	40 352	6	0	34 479	6	0	25 557
7	0	24 811	7	0	21 052	7	0	27 310
8	0	30 145	8	0	21 707	8	0	25 239
9	0	27 973	9	0	21 351	9	0	55 804
10	0	25 019	10	0	21 578	10	0	25 142
11	0	25 748	11	24 353	22 011	11	0	29 629
12	0	29 410	12	0	21 390	12	0	30 251
13	0	26 001	13	0	21 298	13	0	25 123
14	0	24 481	14	0	21 578	14	0	26 226
15	0	24 458	15	0	21 765	15	0	24 459
16	0	33 985	16	0	20 815	16	0	33 772
17	0	24 811	17	0	22 333	17	0	25 166
18	0	25 287	18	0	20 885	18	0	26 855
19	0	28 108	19	0	21 456	19	0	28 478
20	0	25 673	20	0	21 207	20	0	26 063
21	0	24 652	21	0	21 436	21	0	26 119
22	0	24 734	22	0	21 685	22	0	25 402
23	0	29 566	23	0	21 054	23	0	30 413
24	0	38 226	24	0	34 501	24	0	27 031

Appendix C

SlowFib-experiment

Experiment results for the slowfib Fibonacci implementation

GetSlowFib(42) - Run			GetSlowFib(42) - Unit test		
Iteration	Microjoule	Nanoseconds	Iteration	Microjoule	Nanoseconds
0	38 800 133	1 738 258 394	0	23 765 259	1 428 685 376
1	32 367 166	1 717 307 088	1	23 204 348	1 379 607 655
2	27 558 706	1 649 472 436	2	24 998 837	1 440 519 581
3	27 525 869	1 648 119 329	3	23 660 279	1 386 287 754
4	27 515 799	1 644 495 035	4	23 070 681	1 382 039 207
5	27 676 199	1 648 558 547	5	23 073 244	1 380 418 389
6	27 670 766	1 648 726 635	6	23 114 321	1 381 560 860
7	27 673 330	1 647 683 766	7	23 244 813	1 380 325 356
8	27 676 382	1 648 045 034	8	23 191 469	1 380 709 364
9	27 662 283	1 646 801 990	9	23 236 452	1 380 613 419
10	27 809 316	1 650 640 957	10	23 241 884	1 383 178 582
11	27 692 373	1 652 432 008	11	23 198 610	1 384 002 653
12	27 622 793	1 648 091 293	12	22 874 209	1 382 408 957
13	27 773 977	1 648 884 195	13	22 961 733	1 384 335 721
14	27 871 999	1 665 008 013	14	23 002 932	1 383 233 389
15	27 838 429	1 650 521 331	15	23 395 875	1 383 970 460
16	27 696 585	1 648 322 080	16	23 130 006	1 379 670 561
17	27 794 667	1 648 883 139	17	23 043 215	1 381 199 626
18	27 794 241	1 647 605 668	18	23 160 097	1 380 719 474
19	27 871 144	1 648 684 980	19	23 114 260	1 380 424 306
20	27 706 900	1 649 438 081	20	23 198 000	1 383 375 602
21	27 839 223	1 647 779 117	21	23 104 677	1 381 119 437
22	27 786 306	1 644 116 309	22	23 290 224	1 379 516 533
23	27 885 182	1 649 452 411	23	23 178 468	1 381 530 424
24	27 777 517	1 648 060 867	24	23 297 670	1 382 308 590

Appendix D

FastReverse

Results for the Newtonsoft `FastReverse` commit experiment with depths 1, 100 and 10 000.

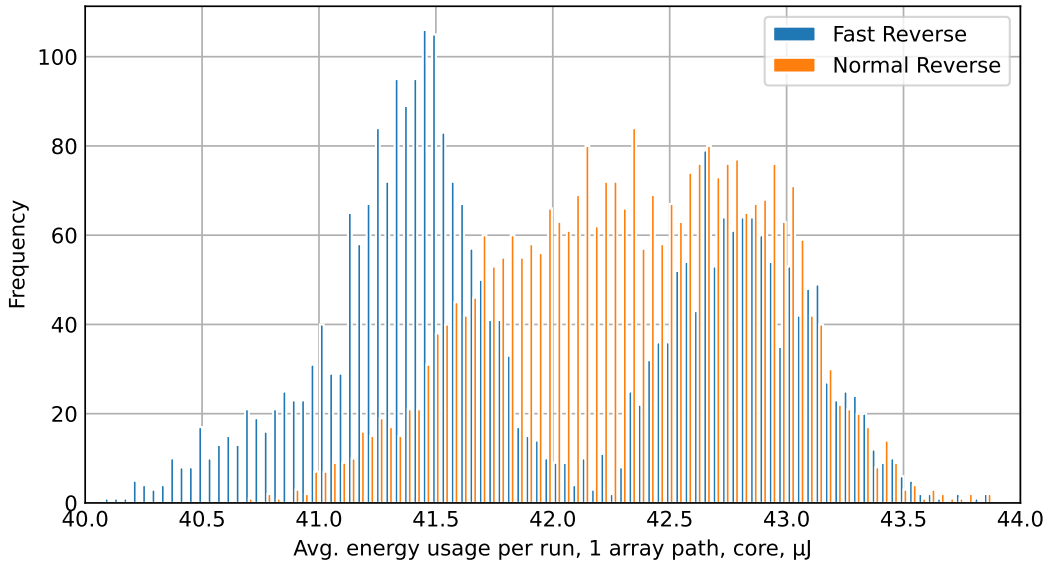

Figure D.1: Core domain with path depth 1, with total number of runs $m_{fast} = 257\,201\,712$, $m_{normal} = 256\,331\,959$, p-value = 1.25e-81

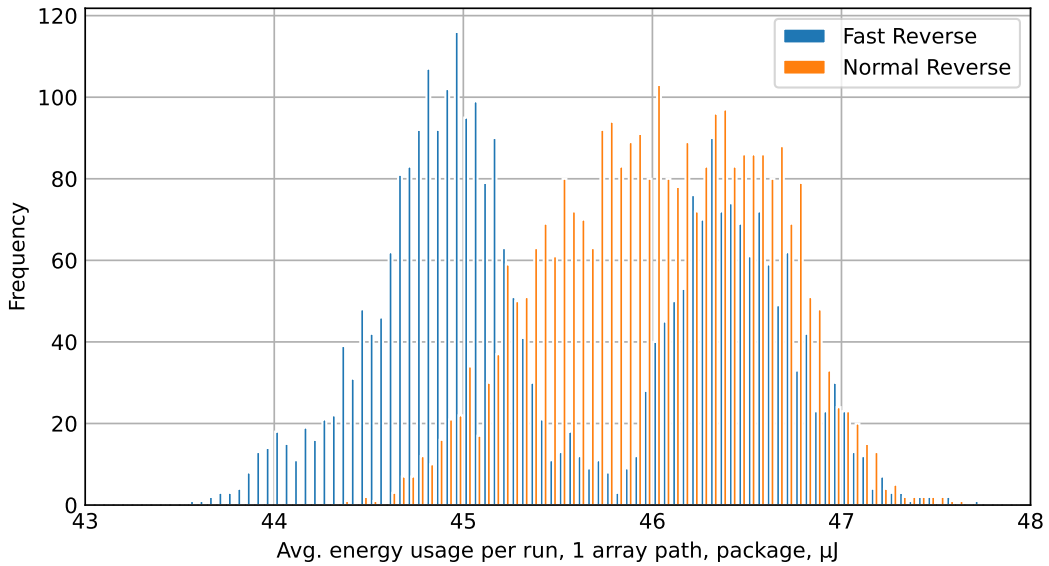

Figure D.2: Package domain with path depth 1, with total number of runs $m_{fast} = 257\,201\,712$, $m_{normal} = 256\,331\,959$,p-value = 4.046e-123

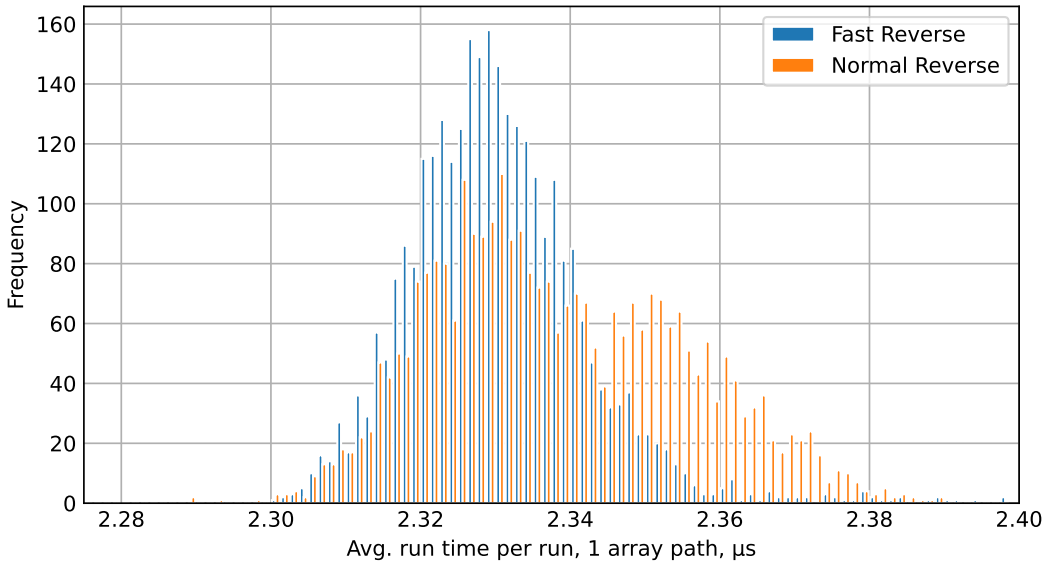

Figure D.3: Time with path depth 1, with total number of runs $m_{fast} = 257\,201\,712, m_{normal} = 256\,331\,959$, p-value = 3.46e-75

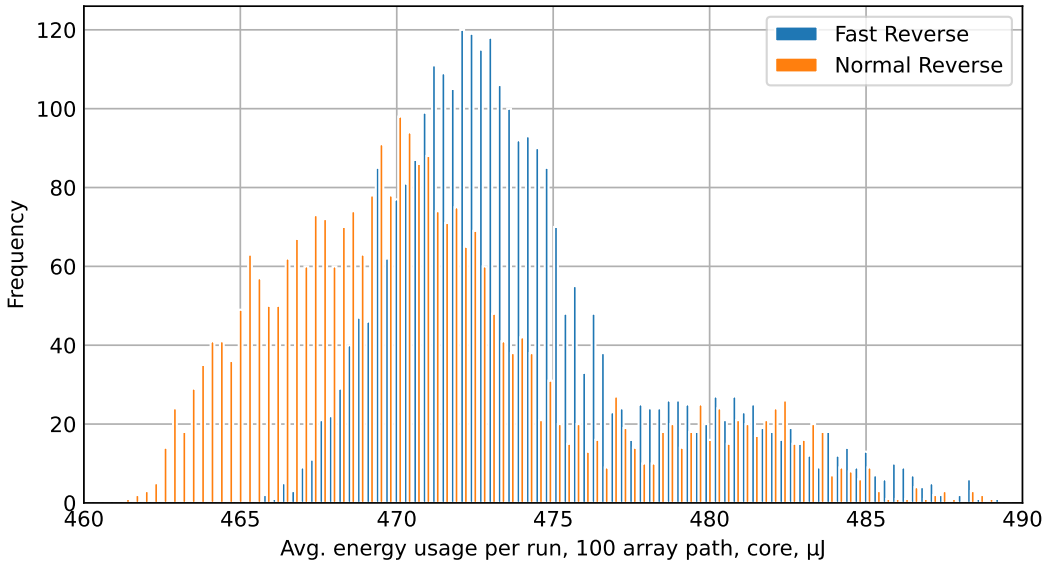

Figure D.4: Core domain with path depth 100, with total number of runs $m_{fast} = 23\,789\,700, m_{normal} = 23\,928\,565$, p-value = 8.03e-151

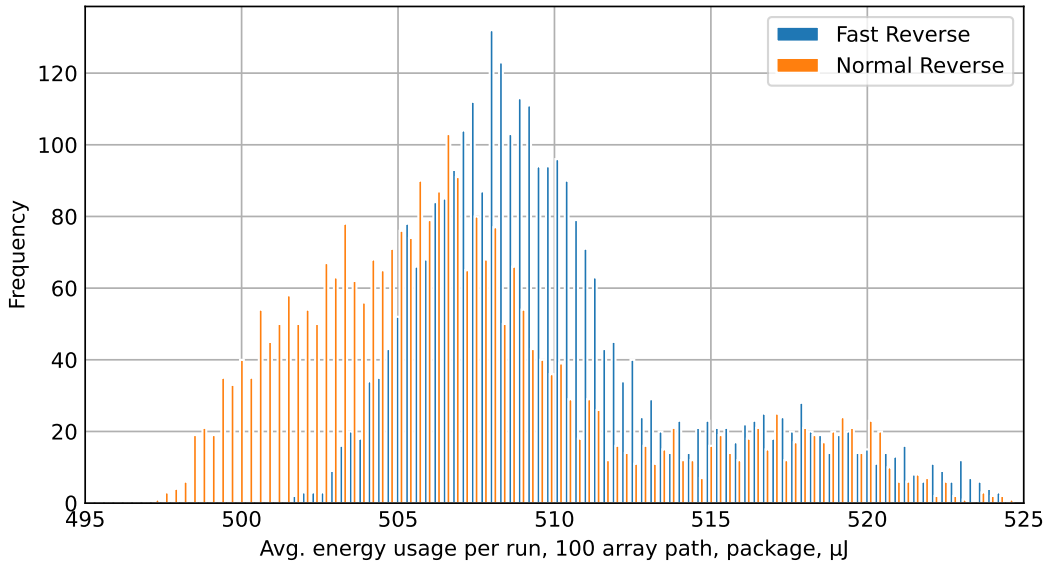

Figure D.5: Package domain with path depth 100, with total number of runs $m_{fast} = 23\,789\,700, m_{normal} = 23\,928\,565$, p-value = 2.42e-144

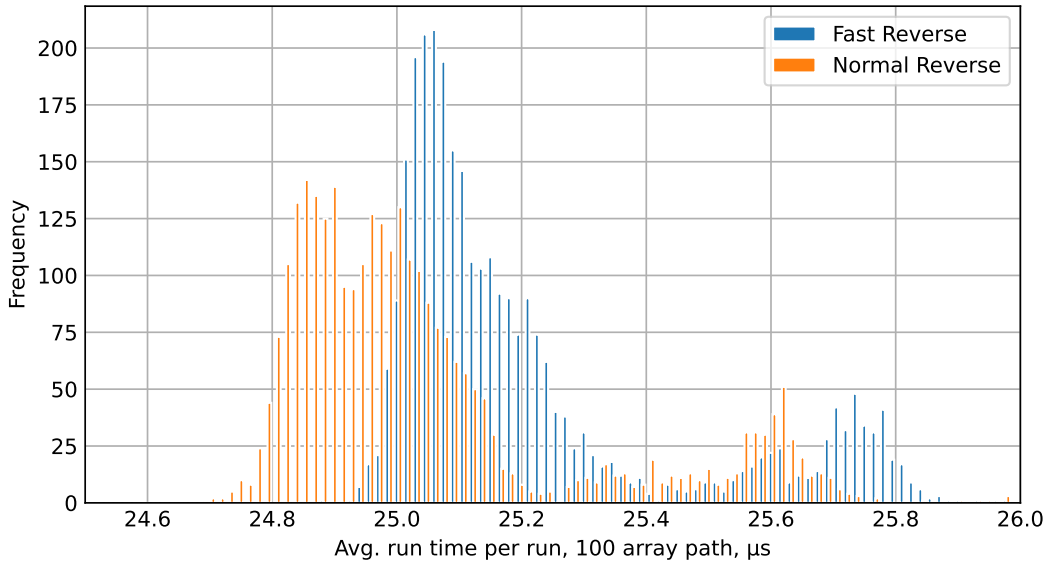

Figure D.6: Time with path depth 100, with total number of runs $m_{fast} = 23\,789\,700, m_{normal} = 23\,928\,565$, p-value = 5.88e-278

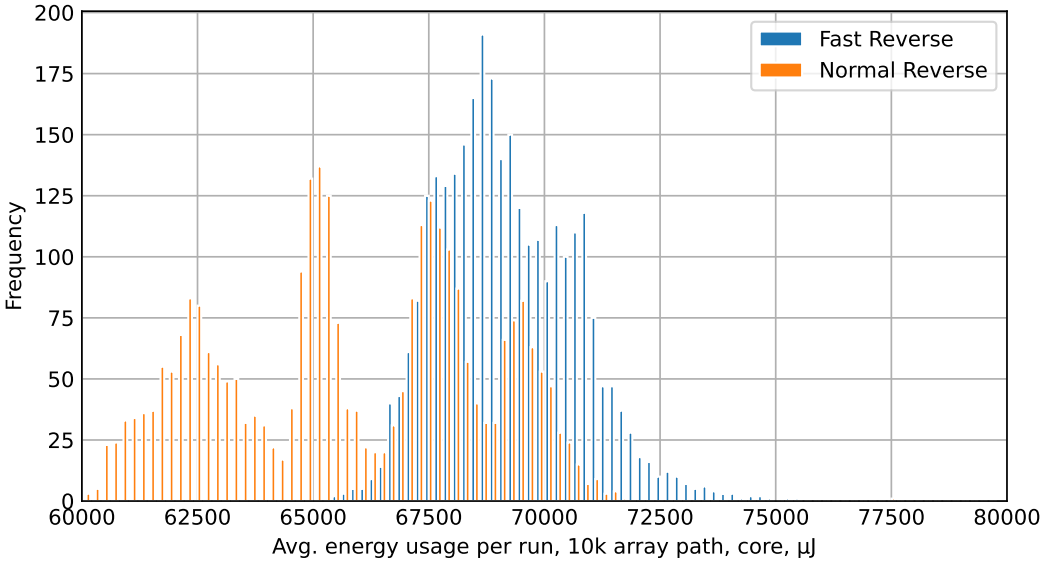

Figure D.7: Core domain with path depth 10 000, with total number of runs $m_{fast} = 143\,291, m_{normal} = 144\,985$, p-value = 0.0

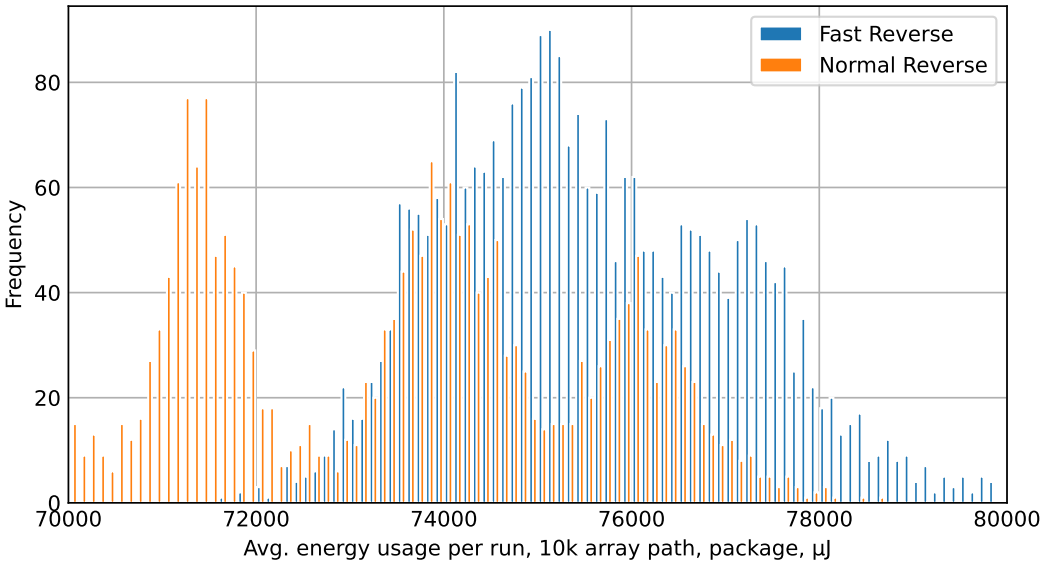

Figure D.8: Package domain with path depth 10 000, with total number of runs $m_{fast} = 143\,291, m_{normal} = 144\,985$, p-value = 0.0

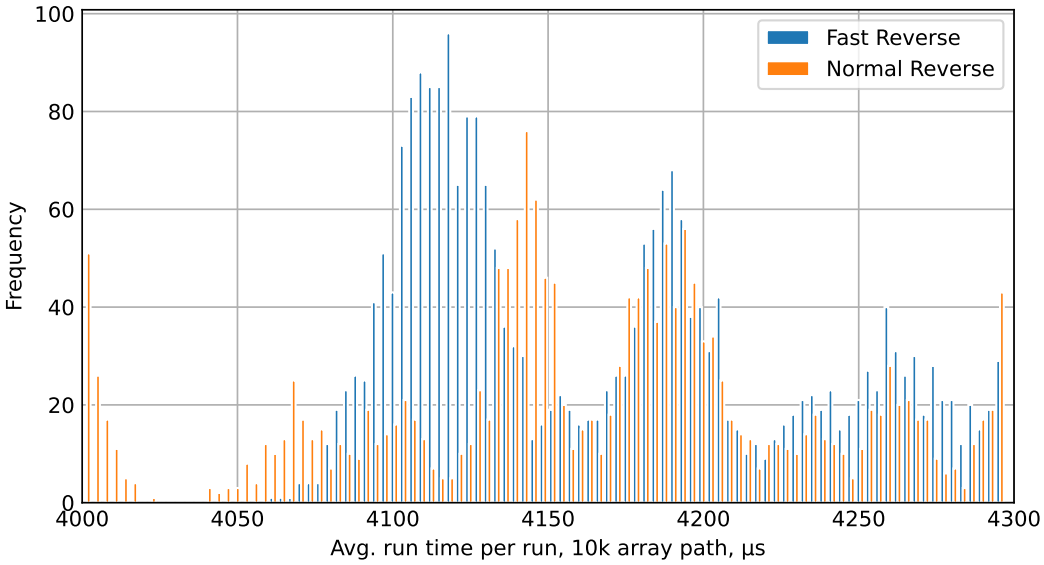

Figure D.9: Time with path depth 10 000, with total number of runs $m_{fast} = 143\,291, m_{normal} = 144\,985$, p-value = 1.61e-32