Thor

A tool for detecting energy hotspots in software

Master Thesis
cs-24-pt-10-01

Aalborg University
Department of Computer Science

Summary

Information and Communications Technology uses an increasing amount of energy. This energy usage can be reduced by optimizing the software that is being used. Focusing on the parts of the software that consume the most energy, also known as hotspots, is a good compromise if the entire software cannot be optimized. Finding these hotspots is essential for this approach.

From this problem and the related works, we formulate the following problem statement.

How can a tool, designed to find energy hotspots, be constructed to help developers gain an insight into a program's energy usage?

This problem statement leads to the following research questions.

- *RQ1: How can hotspots reliably be detected in a program?*
- *RQ2: How does the hotspot detection compare to similar tools?*

In this thesis, the design, implementation, and testing of Thor, a tool for detecting hotspots, is presented. Thor differentiates itself from other tools by not using energy estimation models to estimate the energy consumption. Thor uses static instrumentation to profile the energy consumption of the functions in a program. Thor currently supports programs written in C# and JavaScript, and can be extended to support more programming languages.

The design outlines the different components of Thor. This includes the overall architecture where it is explained how the client interacts with Thor, and how Thor interacts with the process-under-test.

The energy consumption is gathered using Intel's Running Average Power Limit, which utilizes Model-Specific Registers to keep track of the consumed energy. Measurements are gathered using a sampler which regularly retrieves the consumed energy for later use. The static instrumentation adds code which indicates when a function has started and when it has stopped. These start and stop points are then

matched to the sampled energy measurements.

The communication between the client, server, and process-under-test is done using TCP. The process that is to be tested is gathered using Git and is then instrumented using either a source code to Abstract Syntax Tree parser when instrumenting JavaScript, or an Aspect Oriented Programming tool when instrumenting C#. During development, an estimation feature was added to the client. This feature is optional, and the intended use of this functionality is for when a program consists of many small functions, each with a runtime of under one millisecond.

We conduct a test to assess whether Thor can detect hotspots, a test to assess how Thor performs when profiling a multithreaded program, and we test Thor on two implementations of an example web service.

The test to assess Thor's ability to detect hotspots consists of a mix of microbenchmarks where the outcome is known before the test execution. The test was also carried out on an existing method to allow for a comparison. The results of both Thor and the other method are identical for the ranking of the hotspots, with a nearly identical measured energy consumption for the different hotspots. This shows that Thor can find hotspots.

The multithreading test shows that multithreading does introduce some pollution to the measurements. This was enough for hotspots with almost the same energy consumption to switch places. For hotspots with a larger gap between them, the ranking did not change.

The example web service tests shows that Thor is able to profile a program which is more akin to what is found in a production environment.

In conclusion, Thor is able to detect hotspots in programs written in either C# or JavaScript, and it can be extended to support programs written in different programming languages.

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

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Thor: A tool for detecting energy hotspots in software

Theme:
Energy Aware Programming

Project Period:
Spring Semester 2024

Project Group:
cs-24-pt-10-01

Participant(s):
Jakob Zacho Søndergaard
Mads Christian Bruun Nielsen
Villiam Fredrik Jacobsen

Supervisor(s):
Bent Thomsen
Lone Leth Thomsen

Page Numbers: 76

Date of Completion:
June 8, 2024

Abstract:

Previous work have created tools which were able to detect hotspots in software. These tools uses estimation to gather the energy consumption of the different components of a software system. We design and develop a tool, that we call Thor, which does not make use of estimation to gather the energy consumption, but instead uses Intel's Running Average Power Limit to gather the energy consumption. Thor uses static instrumentation and has support for C# and JavaScript with the possibility to extend support to more programming languages. Thor's ability to detect hotspots is tested using two tests. The system used to test Thor, is running the Ubuntu Server operating system. For one of the tests we use an existing method for detecting hotspots, the results from which we use to compare with the results generated by Thor. From the results, we found that, for the first test, Thor was able to detect the hotspots which the existing method found and the reported energy usage was similar. The second test showed Thor in a more complicated scenario where it also detected the hotspots.

Contents

Preface

Acknowledgement

We would like to thank our supervisors Bent Thomsen and Lone Leth Thomsen for their guidance throughout the project and for their constructive feedback.

Usage of Artifical Intelligence

Artificial intelligence has assisted with the development of the implementations in this project by using GitHub Copilot[1]. No artifical intelligence has been used to aid in writing this thesis.

Aalborg University, June 8, 2024

Jakob Zacho Søndergaard
<jsande17@student.aau.dk>

Mads Christian Bruun Nielsen
<mcbn19@student.aau.dk>

Villiam Fredrik Jacobsen
<vjacob19@student.aau.dk>

Chapter 1

Introduction

The energy consumption of Information and Communications Technology keeps increasing[2, 3]. One of the areas where work is being done, to reduce the amount of consumed energy, is with software. While software does not consume energy directly, it controls the hardware which consumes energy. By optimizing software's usage of the hardware, energy consumption can be lowered. One way to reduce the amount of time used optimizing software is to focus on optimizing the parts of the software which uses the most energy, also known as hotspots. Optimizing hotspots would not improve the energy efficiency as much as optimizing all parts of a program, but in a scenario where there is a deadline, it is a compromise between energy efficiency and time. The problem with this approach is finding the parts of a program that uses the most energy. This project explores how to find these parts.

Related works, focused on profiling software, are presented in Chapter 2. It was found that there is a lack of tools or methods able to detect energy hotspots of generic programs while not utilizing energy models to estimate energy consumption.

This leads to the following problem statement.

How can a tool, designed to find energy hotspots, be constructed to help developers gain an insight into a program's energy usage?

This problem statement is divided into the following Research Questions (RQ):

- *RQ1: How can hotspots reliably be detected in a program?*
- *RQ2: How does the hotspot detection compare to similar tools?*

Answering these Research Questions will subsequently answer our Problem Statement.

This project details the development of the hotspot detection tool called Thor. The focus of this project is on the energy profiling of functions. In this project, we use the term function to refer to functions, methods, and procedures. The reason for focusing on functions and not arbitrary pieces of code is to simplify the project. Since functions are used to encapsulate the code of a system, focusing solely on profiling functions still allow for measuring all parts of a program.

This thesis will start with a brief description of the related works in Chapter 2. Then the methodology is presented in Chapter 3, where our design of Thor is detailed. This includes information about the different components and related choices to these components. The implementation of Thor is described in Chapter 4. Here the specific implementation of each component is presented with argumentation for our choices. We use the experimental setup detailed in Chapter 5 to test the implementation. This chapter explains the hardware used for the tests and the different tests conducted. The results from the tests are presented in Chapter 6. In Chapter 7 we discuss the validity of our results and other issues encountered in the project. Lastly, in Chapter 8 we conclude our findings, and we answer the research questions and the problem statement. This chapter ends with presenting proposals for future work.

Chapter 2

Related Works

This chapter presents work related to this project. This includes work which has a focus on the profiling of software.

Jagroep et al.[4] proposed a method for profiling the energy consumption of software, which can be used to detect energy hotspots. Their method uses stubbing, where they stub each functionality that they wish to measure in conjunction with a leave-one-out method. They measure the consumed energy by using a WattsUp Pro power meter[5] and Microsoft Joulemeter[6]. By executing multiple variants of the software, where for each variant, a functionality has been stubbed, they can determine the impact of the stubbed functionality on the energy consumption. They tested the method on parts of a city guide service with a variable amount of visitors. From the test they were able to identify the impact the stubbed functionality had on the energy consumption. This method allows users to measure the energy importance of most functionality but it requires the manual creation of stubs and the execution of the entire software for each functionality to profile.

Rajput et al.[7] developed a tool for adding energy measurements of API calls to Tensorflow into Python scripts using static instrumentation. This tool allowed them to automate usage of their framework, which required instrumentation of the code to enable energy measurements with Intel's Running Average Power Limit (RAPL)[8] and Nvidia System Management Interface[9]. Their developed tool is tailored to energy measurement of deep learning and is tested on Tensorflow tutorials where they concluded that it was able to measure the energy consumption of API calls.

Schubert et al.[10] developed a tool they call eprof for relating energy consumption to code locations. The code locations are retrieved by using stack traces and the energy consumption is estimated using energy models. The tool is able to estimate

the energy consumption of the CPU, memory, and devices such as hard-drives. For the CPU and memory they use statistical profiling by using the hardware performance counters (HPC) programmed to generate an interrupt when certain thresholds are reached. This interrupt is the signal for eprof to begin capturing the stack trace. The HPCs are programmed to match the energy model used such that they can be used to estimate the energy consumed. In a test they performed, their tool incurred a maximum overhead of 2.7%. The highest error they reported for the energy estimation was 7.2%.

Noureddine et al.[11], created a tool, which they call E-Surgeon, for measuring energy consumption at a thread or method level for Java-based applications. This tool is composed of two components. The first is called PowerAPI which collects information about how much energy the process consumes. The second is called Jalen, which is a profiling tool that either uses bytecode instrumentation or injects profiling code into the application code. From this, events related to energy consumption are gathered. This data is then correlated with data about energy consumption using power models. They report that Jalen incur an overhead of 43.34%. They report a margin of error of up to 3% when compared to a powermeter.

Stephenson et al.[12] developed an instrumentation tool for GPUs. This tool uses instrumentation to inject calls to a user-defined function. They show that the tool can be used to analyse a range of things such as memory, resilience, and more. They found that the instrumentation incurred an overhead in the benchmarks they performed. The overhead resulted in a slowdown of up to 160 times the base value.

Ritu Arora et al.[13] performed a comparison of three different metaprogramming approaches. The three approaches are Compile-time metaobjects with OpenJava, Load-time class adaption with Javassist, and Aspect Oriented Programming (AOP) with AspectJ. They use the three approaches to create a profiler, to assess the runtime of functions within a program. They do this by inserting code at the start and end of each function. In their discussion, they highlight different aspects of these approaches. OpenJava is flexible but requires changes to the source code and has a problem with finding return statements. Javassist changes the bytecode at either load or compile time. AspectJ is highlighted as a safer option, as it uses language constructs to define the changes to the code, but the locations to change can be hard to specify.

Thomas Ilsche et al.[14] presents an overview of existing techniques for profiling. They also combine some of these methods to show how different approaches can complement each other. The first combination is a message-passing-interface (MPI) and call-path sampling combination. The combination is developed as a plugin to an existing tool which contain an instrumented library for a MPI. From the results

of this combination, they show that MPI enables a more detailed analysis. The second combination is a combination of Hardware Counter sampling and the instrumentation of function calls and MPI calls. From the results of this combination they show that the function and MPI instrumentation, which occur on enter and exit events, give more information in short functions, whereas the sampler is better for longer running functions where multiple samples can be gathered between the enter and exit events.

Lehr et al.[15] propose the framework PIRA for automatic instrumentation refinement for performance analysis. The framework uses instrumentation to analyse the code and is designed to iteratively refine the instrumentation to include more data about the important regions. Call-graphs and the amount of statements are used to determine which parts contain more work than others. The choice of parts to refine is based on one of two heuristics selection strategies. The GNU profiling interface[16] is used to insert calls at the entry and exit of functions. They evaluate the framework on four benchmarks in regard to the time needed to refine the instrumentation and the resulting instrumentation configuration. In most cases, PIRA converged towards the hotspots while lowering the overhead by not instrumenting less important parts.

From the related works, we identified an area where work could be done. This area is the lack of a tool or method for the detection of energy hotspots in generic programs which does not use energy models to estimate the energy consumption of programs. While Jagroep et al.'s method does not use energy models, their approach is not automated. Therefore the tool developed should be automated. Inspiration can be taken from the performance focused related work on how this can be achieved. Since such a tool or method does not exist, it is unknown whether it is better than the tools and methods which uses energy models, or the existing method which does not use energy models. By developing such a tool or method, an insight into the advantages or disadvantages can be gained.

Chapter 3

Methodology

This chapter explains the design and the thoughts behind the different parts of Thor. Additional technical knowledge of Intel's RAPL is presented, which explains how to use it to measure energy consumption.

3.1 Conceptual Design

This section describes the design of Thor. The purpose of the different parts and components of Thor are presented.

Thor is designed to offload profiling to another machine. This is to allow the developer to continue the development on their own machine without affecting the profiling. If the developer is to access the results of the profiling from their own machine, then Thor would need to be able to act as a server.

By utilizing a server as the machine running the profiling, more than one developer would be able to see the results of a profiling by connecting to the server. This would allow developers to collaboratively analyze the results and plan what parts of the software that should be focused on improving. In this project, RAPL is used for measuring energy consumption. RAPL is explained further in Section 3.3. Accessing RAPL measurements requires administrative privileges. By developing Thor as a server, it means only Thor is required to run with these privileges. Programs that are to be measured, are not required to be granted administrative privileges.

It is still possible for developers to utilize the tool on their own machine, but it comes with the overhead of additional communication layers introduced by the server.

The overheard for each request is small, but with enough requests it can potentially become a problem. This is explored in Section 5.2.3.

3.1.1 Architectural Design

The architecture of Thor is designed as a collection of components. It is designed such that each component can be customized to fit a specific workflow without having to make major changes to other components of Thor.

A diagram of the architecture can be seen in Figure 3.1.

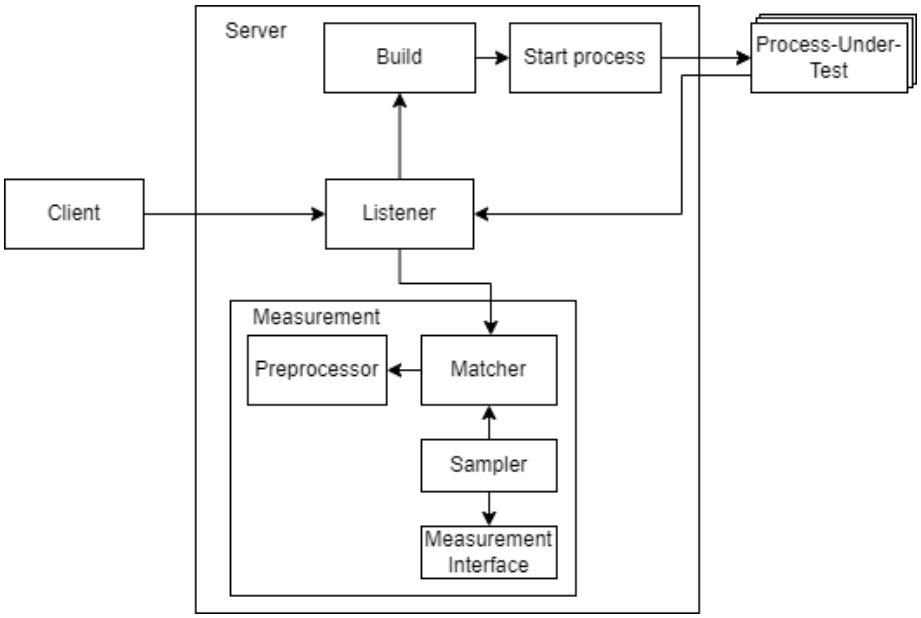


Figure 3.1: Architectural Design. The arrows indicate the direction of which communication is started.

The server consists of the following components:

- `Listener`
- `Build`
- `Start process`
- `Measurement`

The `Listener` component is responsible for handling requests from both the client and the processes-under-test. The communication between the `Client` and the `Listener` is what facilitates the user starting the profiling of a program and retrieving the results. The communication between the processes-under-test and

the `Listener` is used by the processses-under-test to signal when a measurement should be taken, and sent to the `Client`.

The `Build` component is responsible for preparing the source code for execution. The retrieval of the source code can be done in multiple ways and can vary based on the needs of the users. The source code can be retrieved from a remote repository, it can be retrieved directly from the client, or it can be moved using a physical medium. In this project, the `Build` component gathers the source code from a remote repository specified by the client using Git[17].

After gathering the source code, static instrumentation is used to annotate the source code, after which an executable is built.

The `Start process` component handles starting the processes which are to be tested. Since each program can require their own launch commands, making a generic command is not ideal. Instead, the `Client` has to supply the command. The command could be given directly by the client. This would make it easy for a developer to change the command should they need to make changes. It could also be bundled with the source code on the remote repository. This would make sharing the command with other developers easier, but at the cost of having to update the remote repository when wanting to change the command. Since the `Build` component already interfaces with a remote repository, we also interface with the repository for the `Start process` component. Because the `Build` component retrieves data from the remote repository, the command can also be retrieved by the `Build` component, and then given to the `Start process` component.

The `Measurement` component consists of multiple sub-components. A sampling approach is used for retrieving measurements, which is where a measurement is retrieved at a regular interval. This choice is described in Section 3.2. The sub-components are the following:

- `Matcher`
- `Sampler`
- `Measurement Interface`
- `Preprocessor`

The `Matcher` component keeps track of all of the necessary measurements that have been taken. Only recent measurements are saved with the oldest being discarded upon the retrieval of a new measurement. Saving all measurements is possible but would require more storage from the machine running the server. Since only recent measurements are needed, the old measurements can be discarded, and less storage is needed.

When a process tells the `Listener` to take a measurement the `Listener` asks the `Matcher` for a measurement. Because a sampling approach is used, a timestamp is included in the request from the `Listener` to facilitate the matching between the request for a measurement and the sampled measurement.

The `Sampler` is the component which requests a measurement from the `Measurement Interface` component at a regular interval. The `Sampler` gives the measurement to the `Matcher` every time it receives one.

The `Measurement Interface` is the component which retrieves a measurement using RAPL. This component reads the RAPL Model-Specific-Register (MSR) when requested and returns the value to the requester.

The `Preprocessor` component is used to process the values retrieved from the MSR and convert them to joules, the process of which is explained in Section 3.3.

3.1.2 Measuring Process

In order to measure the energy consumption of a section of a program, modifications to the source code are made. The modifications are added with static instrumentation, which is explained in Section 3.1.3.

Since the process-under-test has to signal to the `Listener` that a measurement is to be taken, a functionality to do this is added to the program that the process-under-test is running. This functionality is achieved with the inclusion of calls to one of two functions which have the functionality to send signals to the `Listener`.

The two functions consist of a `start` and a `stop` function. These are inserted in the source code to denote the beginning and the end of the piece of code to be measured. Each function sends a unique identifier with the signal. This signal is used to differentiate between `start` and `stop` calls, but the identifier also allows for the `start` call to be paired with the corresponding `stop` call. This is useful if a function has a different function call within it, as this would cause multiple `start` calls and then multiple `stop` calls.

An example of the `start` and `stop` function calls can be seen in Listing 3.1.

```
1 Framework.start("id")
2 someFunction()
3 Framework.stop("id")
```

Listing 3.1: Start and Stop calls

3.1.3 Static Instrumentation

The measuring process makes use of `start` and `stop` function calls to signal when a measurement is to be taken. This means that the function calls has to be added

to the program.

The instrumentation can be done automatically with either static or dynamic instrumentation. Static instrumentation implies instrumentation before or at compile-time, while dynamic instrumentation is instrumentation at runtime. Dynamic instrumentation works with Just-In-Time (JIT) compiled languages such as C#[18] and Java[19], whereas it might introduce problems with ahead-of-time compiled languages where relevant details such as function names can be compiled away. Static instrumentation will be used for this task, as it is able to utilize all details within the source code. One way to apply the instrumentation statically is to make use of a parser which constructs an Abstract Syntax Tree (AST) from source code. By parsing the source code into an AST, the code can be analyzed and instrumented without needing to account for syntax. Another approach is to use AOP. AOP is a method of applying code after the default compilation step, meaning that it is possible to insert additional code to be executed[20]. This additional code could, for example, be the code that performs the energy measurements.

In this project, both AOP and a source code to AST parser will be used to instrument two different programming languages. The implementation and usage of these is explained in Section 4.2.

3.2 Gathering measurements

There are a couple of different ways to gather energy measurements. One way to retrieve measurements is on demand. This can be done in one of two approaches. The first approach is that the process can send a request to the `Listener` component which afterwards tells the process it has finished taking the measurement. This way guarantees that each request gets the correct measurements, but slows down the process. The second approach is that the process does not wait for a response from the `Listener` component. This approach is faster for the process but depending on the performance of the `Listener` component, it could report incorrect measurements.

A second way is to regularly retrieve measurements with a sampler and then, when a request for a measurement is retrieved, match the request to a measurement. The matching can be done with a timestamp if the sampler saves the timestamp for each measurement, and if the process sends a timestamp with the request for a measurement. Since a timestamp is given with each request, and each measurement has an associated timestamp, the moment the two are matched is not as time sensitive as if the process was waiting for a response. This allows the server to perform the matching even after the process is finished, lessening the overhead it incurs, which reduces the impact it has on the performance of the system. Another

aspect of a sampler is that it introduces an overhead with lesser variance[14].

It is possible that a request for a measurement lies between two different sampled measurements. This can be handled in a couple of ways. The chosen measurement can be rounded up or down to the nearest sampled measurement. This approach has some problems. If both the `start` and `stop` requests get rounded to the same sampled measurement, then the used energy will be zero. This means that even if that specific function were to use most of the power due to it being executed significantly more than the other functions, the results would not show it as a hotspot. The chosen measurement can also be an estimate. Since it is possible for the matcher to wait until more sampled measurements have been taken, the energy consumption between two samples can be estimated. Since each request and sample have an associated timestamp, the request can be matched to the estimated point between the samples.

In this project, the requests are matched to the nearest measurement, but the client has an optional estimation feature, which is explained in Section 4.5.

3.3 Intel's Running Average Power Limit

Intel's Running Average Power Limit (RAPL) is an interface that allows retrieval of energy measurements through Model-Specific-Registers (MSR).

The client hardware has support for:

- *Package*, which is the processor die[21, Sec 15.10].
- *PP0 (Power Plane 0)*, which refers to the processor cores[21, Sec 15.10.4].
- *PP1 (Power Plane 1)*, which refers to a specific device in the uncore, such as an uncore graphic device[21, Sec 15.10.4]

The server hardware has support for:

- *Package*[21, Sec 15.10].
- *PP0*[21, Sec 15.10].
- *DRAM (Dynamic Random Access memory)*[21, Sec 15.10].

The MSR's are updated every 976 microseconds by default[21, Sec 15.10.1]. Intel's CPUs have supported this interface since the Sandy Bridge architecture, and AMD's CPUs have supported it since the release of the Zen architecture[22].

Additionally, the MSR_RAPL_POWER_UNIT MSR is used. It specifies how the values read from the other registers can be converted. Contained from bits 8 to 12 in the MSR_RAPL_POWER_UNIT register, is the Energy Status Unit (ESU), which

specifies the unit for converting the values from the other registers to joules, which can be done using the following formula: 0.5^{ESU}[21, Sec 15.10.1].

An attempt was made to try to modify the MSR_RAPL_POWER_UNIT MSR register. However, it was discovered that this MSR register is read-only. Therefore, it is not possible to increase the speed of the RAPL measurements.

Chapter 4

Implementation

This chapter details the implementation of Thor. It describes the various choices made and some of the important code that was developed as part of the solution. The full implementation can be found on the associated GitHub Organization[1].

4.1 Server Components

This section details the implementation of the components within the server. The components are described in Section 3.1.1. The implementation of the server can be found on the associated GitHub repository[2].

A goal for the development of the server is a low overhead, such that the server does not use resources needed for the process-under-test. Because of the need for a low overhead, the server is written in the Rust programming language. Other languages such as C and C++ could have been used, as they also are efficient languages[23].

4.1.1 Listener

The implementation of the `Listener` component is split into two threads, the `listening thread` which is responsible for listening, and the `responding thread` which sends measurements to the client. This split allows the server to listen for `start` and `stop` calls and send measurements to the client simultaneously. Inspiration has been taken from the bounded buffer problem also known as the producer-consumer problem[24, Sec 4.3]. This is because both threads share a queue, where

[1] https://github.com/cs-24-pt-10-01
[2] https://github.com/cs-24-pt-10-01/thor

the `listening thread` enqueues start and stop calls, and the `responding thread` dequeues these calls to send measurements to the client. The queue removes a potential bottleneck for the `listening thread`, as it is not dependent on how fast the `responding thread` can send measurements to the client.

The methods of communication used between the server, client, and process-under-test is an important choice, as it has an impact on the performance of the system due to the added I/O. It is possible to communicate between processes on the same machine using, for example, Inter-Process Communication (IPC). Another option is network sockets, which are intended for streaming data to other systems. IPC can only be used within the same machine, which means that IPC cannot be used to communicate with the client, unless the client is on the same machine. As such, network sockets are used between the client and the server. For packets sent between the processes and the server, it is possible to use either IPC or network sockets, since the processes are located on the same system.

To choose between the two categories, the performance of the different methods are considered. The performance of Anonymous Pipes, Named Pipes, UNIX Sockets, and TCP sockets has been benchmarked[25]. The first three are related to IPC, and TCP sockets are a type of network socket. It was generally found that Anonymous Pipes and Named Pipes are good choices when it comes to small packets. However, as the buffer size increased, UNIX sockets and TCP came out on top. The performance comparison between UNIX sockets and TCP sockets is largely negligible. Therefore, network sockets are used. Aside from TCP sockets, there are also UDP sockets. While UDP is faster than TCP[26], it has the possibility of losing packets. These lost packets which represents energy measurements, can give an incomplete picture of the energy consumption to the developer. Therefore the choice was to go with TCP sockets. An example of a packet sent by the process-under-test, can be seen in Appendix E.

Data transfer

One of the potential issues that can be encountered, is the amount of data that is collected and subsequently transferred to the client. Depending on how much work each function in a program does, the amount of data that is sent in a given timeframe changes.

The format that is used to transfer data from the server to the client is JavaScript Object Notation (JSON). JSON was chosen as it is a commonly used medium for sending data over a network.

Since the act of sending data to the client has no impact on Thor's ability to find hotspots, using a different medium will not affect the results, but it may affect the time that is required to transfer the data.

An example of a JSON packet can be found in Appendix E.

4.1.2 Measurements

In the implementation of Thor, the energy consumption is measured with Intel's RAPL, which was found to be as accurate as measurements at the plug[27]. Measurements from the interface have a resolution of around 1 kHz, which has been shown capable of capturing the effect of events, such as JIT compilation, in contrast to the 1 Hz from a power plug which was not sufficient for the task of showing the effect of JIT compilation[28].

The RAPL measurements can be read on the Windows operating system (OS) using the `rdmsr` kernel command, while on the Linux OS it is possible to read the measurements from the MSR using the filesystem, or by calling the `rdmsr` kernel command. In this project we test on the Linux OS and access measurements using the filesystem.

As mentioned in Section 3.1.1, energy measurements are retrieved at regular intervals. The amount of time between the intervals is set within a configuration file to enable the user to define the granularity wanted. The granularity is limited by the resolution of RAPL, so an interval under 1 kHz will only yield additional information about when the MSR is updated. This information is useful as the ideal times to sample are the when the MSR updates. Since the sampler is independent from the MSRs being updated, a higher sampling rate results in some samples being closer to the ideal times. The smaller the sampling interval is, the more overhead the sampler will incur.

Similar to the implementation of the listener, the implementation of the measurement components also takes inspiration from the bounded buffer problem. The task of sampling at regular intervals is given to a single thread, that produces measurements to a queue. The measurements within the queue are consumed by function calls to the matcher executed by the responding thread of the listener component. This setup allows for sampling without disturbances that can result in delays between samples such as sending measurements to the client.

The matcher component is implemented as a function that uses a range map to match measurements from the queue with timestamps from the `start` and `stop` calls. The range map is a data structure that maps values onto ranges[29], in this case, the values are measurements and the ranges are time. The implementation of the range map is based on a B-tree which allows for access, insertions, and deletion in O(log(N)) time, where N is the number of measurements[30]. The range map allows for efficient matching between timestamps and measurements, but it has to be kept updated with new measurements before usage. Because

of the cost of updating the range map, the matcher supports matching multiple timestamps simultaneously, as this allows for updating the range map once for multiple timestamps. Updating the range map includes adding measurements from the queue and removing old ones. The removal is based upon a variable declaring how long a measurement will be stored, this variable affects performance as the access time of the range map is dependent on its size.

It can be argued that the variable should be low, however, if too many measurements are removed, then matching timestamps can be impossible because of missing measurements. The value should be set in regard to the expected maximum delay between the process-under-test and the server.

4.1.3 Build & Start process

The `Build` and `Start process` components have been implemented as a single component. This increases the coupling between the two parts, but simplifies the implementation, as the build script and the run script can be combined.

The implemented joined component uses Git[17] to clone repositories using a link supplied by the client. The repository is expected to contain a run script, which is used to specify how the repository should be instrumented, built, and run. Section 4.2 presents the tools that are used for the instrumentation.

The run script is executed by a thread, which is started by the `Listener`, using a system call. This means that when the process is finished, the thread can disconnect the client indicating that the profiling is finished.

The two parts can be separated into different components by moving each of their parts of the shared code into their own part, and each of their parts of the run script into two separate scripts.

4.2 Static Instrumentation

In order to gather information about the executed code, custom code is inserted. The techniques and tools used for instrumenting code depends on the language that is being targeted.

As mentioned in Section 3.1.3, we use a source code to AST parser and an AOP tool.

Using a source code to AST parser for the purpose of instrumenting the code requires rules for modification to be created. These rules could include that it has to target function nodes and insert new nodes before and after. Source code to AST parsers have the benefit that changing the target node is simple. This means that

using such a parser allows for instrumentation of more than just functions, such as different types of loops. Since the focus of this project is to profile functions, this functionality is not needed for this project.

Using AOP requires the developer to define aspects, which are the functionality to be added to the code[20]. In our case, the aspects are calls to the shared library for measurements. Where to insert the aspects into the source code has to be defined. This can be done through inserting tags or attributes within the code, or defining composition rules.

4.2.1 Instrumenting JavaScript

Using the Acorn parser[31] we have implemented a tool for instrumenting JavaScript code[3]. Acorn is used to parse the given JavaScript code into an AST, which is then walked through using a tree walker.

The AST is instrumented one block at a time, where each line within the block is searched for expressions containing function calls. If a function is found within a line, then the line is wrapped within a start and stop node.

Return statements are handled differently to ensure that the stop node is reached. If a function call is found within an expression of a return statement, as seen in Listing 4.1 on line 5, then the returned expression is put into a constant declaration instead, as seen in Listing 4.2 on line 10. This is because the stop node is placed after the function call, so if the function call is in a return expression, then the stop node would be placed after the return expression and therefore never be reached. A new return statement is then added to return the constant.

To enable execution of the code within the instrumented AST, a library called `estree-util-to-js` is used to convert the AST to JavaScript code[32]. An example of code before and after instrumentation can be seen in Listing 4.1 and 4.2.

```
1 functionCall();
2 const variableWithFunc = example();
3
4 function example() {
5     return func();
6 }
```

Listing 4.1: Example code before instrumentation

```
1 const rapl = require('./rapl.js');
2 rapl.start("1:example.js:functionCall");
3 functionCall();
4 rapl.stop("1:example.js:functionCall");
5 rapl.start("2:example.js:example");
```

[3] https://github.com/cs-24-pt-10-01/Using-acorn-to-decorate-JS

```
const variableWithFunc = example();
rapl.stop("2:example.js:example");
function example() {
  rapl.start("5:example.js:func");
  const __result = func();
  rapl.stop("5:example.js:func");
  return __result;
}
```

Listing 4.2: Example code after instrumentation

The instrumented code uses a file called `rapl.js`, which is a script that contains start and stop calls for the shared library explained in Section 4.3.

4.2.2 Instrumenting C#

To instrument C# code, the AOP tool called Metalama[33] is used. This tool allows for the insertion of aspects at compile-time.

The aspect to be inserted into the code is defined with the `LogAttribute` class. This class contains a method called `OverrideMethod` that is used to override methods with calls to the shared library. The implementation of this class can be seen in Listing 4.3. The `LogAttribute` class is applied using the `Fabric` class, which defines where to apply the aspect at compile time. The `Fabric` class can be seen in Listing 4.4.

```
public class LogAttribute : OverrideMethodAspect
{
    public override dynamic? OverrideMethod()
    {
        var methodInfo = meta.Target.Method;
        var declaringType = methodInfo.DeclaringType;
        var methodName = methodInfo.Name;
        var namespaceName = declaringType?.Namespace;

        Thor.Thor.start_rapl($"{namespaceName}.{declaringType?.Name}.{
    methodName}");
        var result = meta.Proceed();
        Thor.Thor.stop_rapl($"{namespaceName}.{declaringType?.Name}.{
    methodName}");

        return result;
    }
}
```

Listing 4.3: C# LogAttribute

```
internal class Fabric : ProjectFabric
{
    public override void AmendProject(IProjectAmender amender) =>
```

```
4         amender.Outbound
5             .SelectMany(compilation => compilation.AllTypes)
6             .SelectMany(type => type.AllMethods)
7             .Where(method => method.BelongsToCurrentProject)
8             .AddAspectIfEligible<LogAttribute>();
9 }
```

Listing 4.4: C# Fabric class

In order to call the `start_rapl` and `stop_rapl` functions, the functions are imported from the shared library under the Thor namespace. The Thor namespace can be seen in Listing 4.5.

```
1 namespace Thor
2 {
3     public class Thor
4     {
5         [DllImport("libthor_lib_sync.so")]
6         public static extern void start_rapl([MarshalAs(UnmanagedType.
    LPUTF8Str)] string lpString);
7 
8         [DllImport("libthor_lib_sync.so")]
9         public static extern void stop_rapl([MarshalAs(UnmanagedType.
    LPUTF8Str)] string lpString);
10     }
11 }
```

Listing 4.5: C# Thor namespace

4.3 Shared Library

To send measurements to the Thor server, a shared library is utilized by the processes-under-test. Using a shared library enables usage in multiple programming languages with a single implementation. The shared library uses the C Foreign Function Interface (FFI), which means that every programming language that supports this interface can use the library. Interfacing with the shared library incur different energy efficiency overheads, but the overhead is not significant[28].

The implementation of the shared library is designed around the `start` and `stop` functions. Both of these functions takes a string identifier as parameter which is used to identify the function that is to be profiled, and sends a network packet to the server. The implementation of the shared library can be found on the associated GitHub repository[4].

Since the shared library is loaded by the process-under-test, it is possible for the process to call the associated `start` and `stop` functions using, for example, one

[4]https://github.com/cs-24-pt-10-01/thor/tree/main/crates/shared-lib-sync

thread, or several threads. To prevent race conditions if multiple threads executes the functions at the same time, synchronization is added to the stream that sends packets with the help of a mutex.

In terms of sending the packets, it is possible to do so on either the executing thread or a background thread. The potential benefit of using a background thread is that the executing threads may be able to offload the work to the background thread, letting them return to their work quicker. However the issue with using a background thread is that it writes the packets in predetermined intervals, causing measurements to be lost if the writing has not finished before the process exits. This can occur with, for example, an unexpected exit or a process crash. It was chosen in this project to perform packet sending on the executing thread, to ensure that all packets will be sent to the server.

The `start_rapl` and `stop_rapl` functions can be seen in Listing 4.6. These functions are unsafe because they are exposed to C pointers. Both functions result in calls to safe functions of the communication library after converting the input into a string. This can be seen on line 12 and 23. The `#[no_mangle]` attribute, seen on line 7 and 18, is necessary to ensure the Rust compiler maintains the function's name when compiled, otherwise, the function's name will be mangled, making it different from the intended name.

```
1  use crate::library;
2  use std::ffi::{c_char, CStr};
3
4  /// # Safety
5  ///
6  /// This function is unsafe because it dereferences the ‘id‘ pointer.
7  #[no_mangle]
8  pub unsafe extern "C" fn start_rapl(id: *const c_char) {
9      let id_cstr = CStr::from_ptr(id);
10     let id_string = String::from_utf8_lossy(id_cstr.to_bytes()).
       to_string();
11
12     library::start_rapl(id_string);
13 }
14
15 /// # Safety
16 ///
17 /// This function is unsafe because it dereferences the ‘id‘ pointer.
18 #[no_mangle]
19 pub unsafe extern "C" fn stop_rapl(id: *const c_char) {
20     let id_cstr = CStr::from_ptr(id);
21     let id_string = String::from_utf8_lossy(id_cstr.to_bytes()).
       to_string();
22
23     library::stop_rapl(id_string);
```

```
24 }
```

Listing 4.6: The `start_rapl` and `stop_rapl` functions as exposed through the FFI.

Listing 4.7 shows the `start_rapl` function from the communication library. The function constructs a packet to be sent, and passes it to the `send_packet` function. The `stop_rapl` function is nearly identical to the `start_rapl` function, and functions identically.

```
1  pub fn start_rapl(id: impl AsRef<str>) {
2      let packet = ProcessUnderTestPacket {
3          id: id.as_ref().to_string(),
4          process_id: process::id(),
5          thread_id: thread_id::get(),
6          operation: ProcessUnderTestPacketOperation::Start,
7          timestamp: SystemTime::now()
8              .duration_since(std::time::UNIX_EPOCH)
9              .unwrap()
10             .as_nanos(),
11     };
12
13     send_packet(packet);
14 }
```

Listing 4.7: Implementation of `start_rapl` in the client module, showing the construction of the packet and the packet being passed to the `send_packet` function.

The `send_packet` function, as seen in Listing 4.8, serializes the packet and sends the packet to the server.

Since there is no server connection on the first call, it will perform initialization on the first execution, seen on line 2, signifying its connection type as a `ProcessUnderTest` on line 7.

After the initialization, the packet is serialized using the `bincode` Rust package to turn the packet into bytes. The serialization happens before the global mutex is locked, as to not perform extra CPU operations while the lock is held.

Once serialization of the packet has finished on line 13, the global mutex is acquired and locked on line 15. The length of the packet is then written to the stream, followed by the packet's data. This can be seen on lines 19 and 20 respectively. The reason behind sending the length of the packet is to tell the server how much data it should expect to read. The size of the packet from the process-under-test can vary due to the identifier parameter, making the packet's size dynamic. If the identifier parameter did not exist, the size of the packet would always be constant. The serialized packet with an empty identifier is 40 bytes in size, and each additional character in the identifier increases the packet size by one byte.

```rust
fn send_packet(packet: ProcessUnderTestPacket) {
    STREAM_INIT.call_once( {
        // making connection
        let mut connection = TcpStream::connect(ADDRESS).unwrap();

        connection
            .write_all(&[ConnectionType::ProcessUnderTest as u8])
            .unwrap();

        *CONNECTION.lock().unwrap() = Some(connection);
    });

    let serialized = bincode::serialize(&packet).unwrap();

    let mut connection_lock = CONNECTION.lock().unwrap();
    let stream = connection_lock.as_mut().unwrap();

    // Write length and then the serialized packet
    stream.write_all(&[(serialized.len() as u8)]).unwrap();
    stream.write_all(&serialized).unwrap();
}
```

Listing 4.8: The `send_packet` function, showing the initialization, serialization, and sending of the packet

4.4 Client

This section will explain the overall idea of the client implementation. The complete implementation can be found on the associated GitHub repository[5].

As mentioned in Section 4.1.2, the measurements take the form of the amount of energy consumed before the function is executed and the amount of energy after it is executed. While this data on its own can be used to gather the necessary information, having an interface to extract the information is beneficial as it can automate away the steps a developer has to take in order to manually extract the information. On an architectural level, as shown in Section 3.1.1, this interface is the client.

We implemented the client as an extension for Visual Studio Code (VSCode)[34]. VSCode is a code editor, that developers can use to write software. Implementing the client as an extension to a tool that the developer is already using, means that they do not have to open different programs to use the client. VSCode has existing functionality to interact with Git[35]. Since our build module, which is detailed in Section 4.1.3, uses Git, it means that everything needed to profile a program to find the hotspots can be done within the same application.

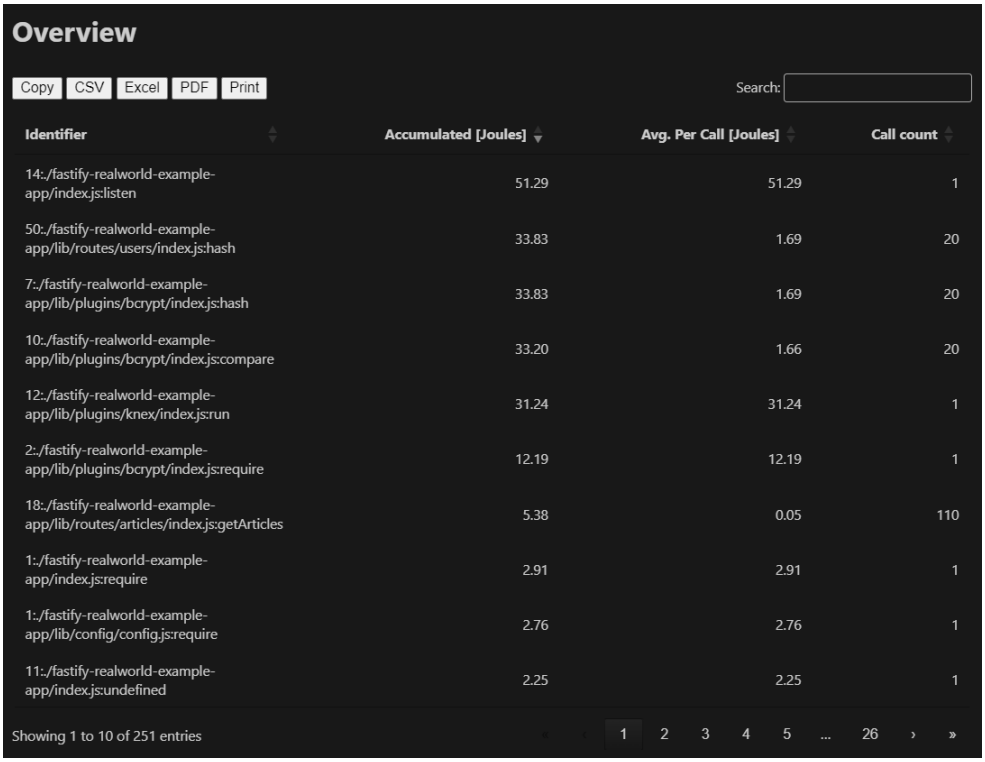
[5]https://github.com/cs-24-pt-10-01/Thor-client

The client has been implemented as a Webview extension[36]. This type of extension uses HTML in addition to JavaScript for the functionality. Node.js is used as the runtime for the JavaScript.

As described in Section 4.1.1, Thor uses a TCP socket for sending and retrieving data from the client. To support this, the client utilizes the `net` module from Node.js.

The JSON received from the server can contain an arbitrary amount of measurements. The server sends the data to the client in chunks of measurements. These chunks are then, during transmission, further divided into smaller chunks by the `net` module. The client cannot discern between the packages it receives so they are all treated equally. This becomes a problem as the collection of measurements has to be reassembled by the client, but the client is not told by the server when the last part of a chunk is received. To signal to the client that the last part has been sent, a delimiter is added. When the client receives a package with the delimiter, it can then identify a complete chunk. Each chunk of measurements is then sequentially given to the part that handles the user interface.

The received data then has to be presented to the user. The presentation includes the accumulated energy consumption in joules, the average energy consumption per function call in joules, the amount of function calls, and a graph. An example of a graph can be seen in Figure 4.2, which shows how the energy consumption per call changes. The accumulated energy is included because it shows the function that uses the most energy overall. The average energy consumed per function call is included to allow for distinction between a few function calls that use a lot of energy and a lot of function calls that use a little energy. This information is presented in a table, an example can be seen in Figure 4.1, and additionally the information is also presented with each graph.



Overview

Copy CSV Excel PDF Print Search:

Identifier	Accumulated [Joules]	Avg. Per Call [Joules]	Call count
14:./fastify-realworld-example-app/index.js:listen	51.29	51.29	1
50:./fastify-realworld-example-app/lib/routes/users/index.js:hash	33.83	1.69	20
7:./fastify-realworld-example-app/lib/plugins/bcrypt/index.js:hash	33.83	1.69	20
10:./fastify-realworld-example-app/lib/plugins/bcrypt/index.js:compare	33.20	1.66	20
12:./fastify-realworld-example-app/lib/plugins/knex/index.js:run	31.24	31.24	1
2:./fastify-realworld-example-app/lib/plugins/bcrypt/index.js:require	12.19	12.19	1
18:./fastify-realworld-example-app/lib/routes/articles/index.js:getArticles	5.38	0.05	110
1:./fastify-realworld-example-app/index.js:require	2.91	2.91	1
1:./fastify-realworld-example-app/lib/config/config.js:require	2.76	2.76	1
11:./fastify-realworld-example-app/index.js:undefined	2.25	2.25	1

Showing 1 to 10 of 251 entries « ‹ 1 2 3 4 5 ... 26 › »

Figure 4.1: Example of the overview table

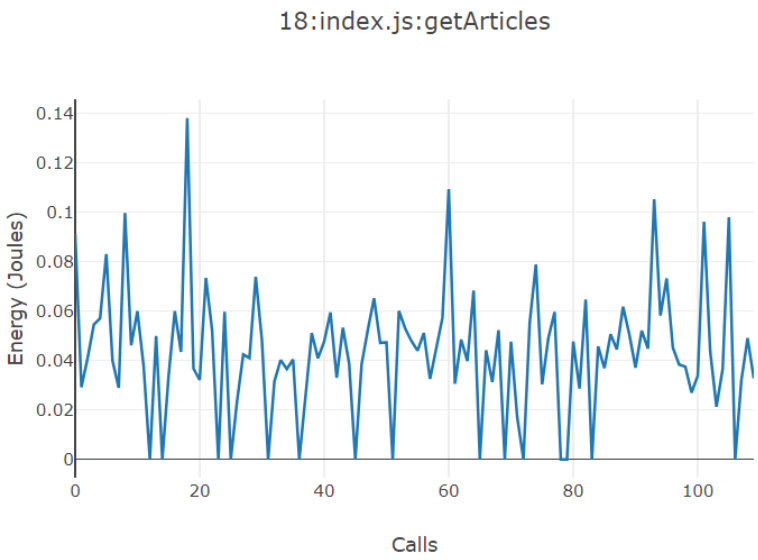


Figure 4.2: Example of a graph produced by the client

One of the limitations of Intel's RAPL is the update rate of the registers. It is possible for one or more functions to finish executions before the register updates. This will cause Thor to report the energy consumption of those functions as zero, as can be seen for some of the calls in Figure 4.2. This problem is discussed further in Section 7.4.

An optional feature of the client is the ability to enable estimation of values which would otherwise have been set to zero. The estimation feature is explained in Section 4.5. Since it is an estimation, the feature is not enabled by default, but it can be enabled in the user interface.

A different approach could have been taken to gather the energy consumption of the fast functions, without the use of estimation. It could have been done by repeating the functions using a loop until the registers update. The register value can then be divided by the number of times the function was executed to get the energy consumption of a single execution. Since this approach is not using estimation, the results more accurately represents the function's energy consumption. There are drawbacks to this approach, which is why we choose not to use it. Firstly, some functions cannot easily be repeated in a loop. Functions that interface with databases are a good example. The same information cannot necessarily be entered into the same database multiple times as it would cause conflicts with already existing data. Secondly, if the program that is being profiled is written in a JIT compiled language, then executing a function in a loop would change the behavior of the program, and the measured energy consumption would no longer be representative of the function's energy consumption in a normal execution of the program. Thirdly, the execution time would increase. Whether the increase in execution time is significant depends on the program that is being profiled.

4.5 Energy estimation

The energy consumption of a function is the difference in the register value between the start and stop call. As mentioned in Section 3.3, the register updates every 976 microseconds. It is possible for functions to be fast enough for both the start and stop to happen before the register updates. This will result in the reported energy consumption of the function to be zero. Since the function does incur energy consumption, the result is misleading. To alleviate this, the energy consumed by the function can be estimated.

The idea of the estimation functionality is that with two sequential measurements from RAPL, the values between them can be estimated. For the estimation, the change in the energy consumed is considered to be linear. While time and energy consumption are not linearly correlated[23], we consider it close enough to be

usable.

The estimation works in chunks of data as it needs two different values to be able to estimate. A chunk consists of multiple measurements where the last measurement has a different energy consumption than the previous measurements.

The estimation needs the energy consumption from the first and last elements, and the timestamp from each element. By assuming a linear correlation between the first and last measurement, the estimated energy consumption for those in-between is a percentage of the measured value change. The percentage can be found for each element by calculating the percentage of the time they use, and then use that percentage to get the energy consumption.

The calculation of the percentage is done iteratively. This means that when the current measurement has been estimated, the part between the starting measurement, and the current measurement is removed, and the current measurement becomes the new starting measurement. This then repeats until each measurement has been estimated. This can be done because of the linear estimation.

Figure 4.3 and 4.4 demonstrate how the estimation works. In Figure 4.3, it can be seen that the current measurement is at 5 time units, or 50% of the time period, so the estimation is therefore that it uses 50% of the energy which is 5 energy units. In Figure 4.4, it can be seen that the new starting position has been moved to the end of the previously estimated part. From here the current measurement gets estimated where it uses 1 time unit, or 20% of the remaining energy, which equates to 1 energy unit. This process repeats until all of the remaining energy has been used.

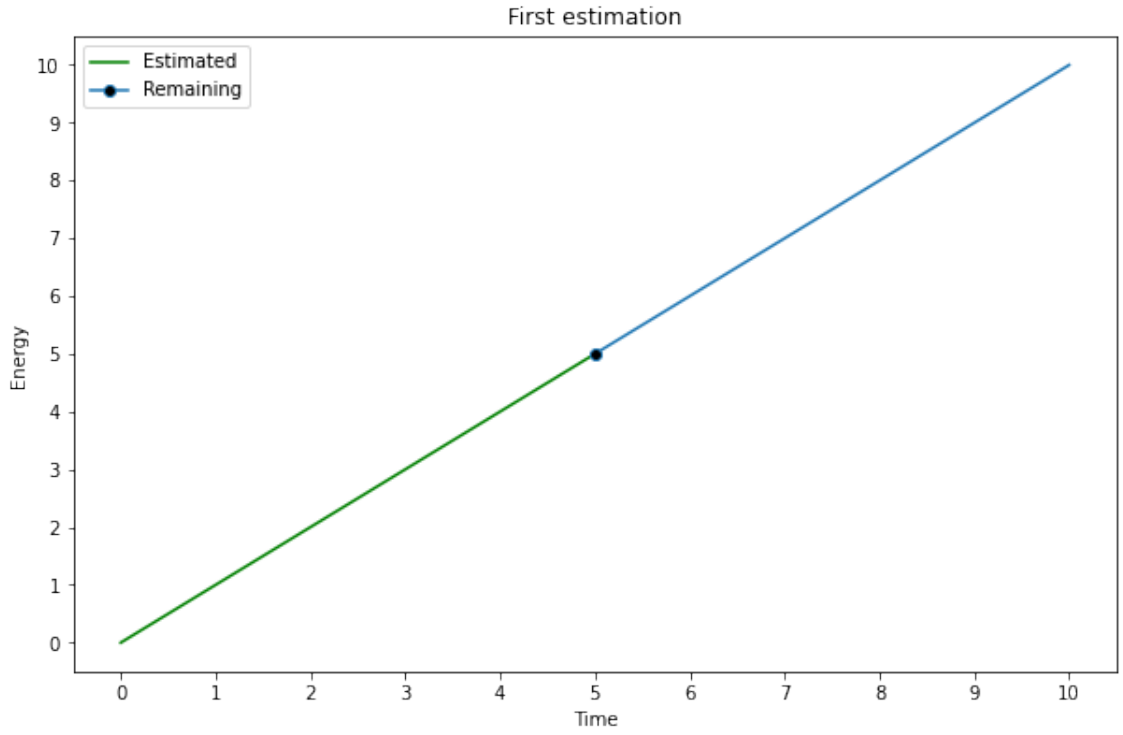


Figure 4.3: Example of first estimation

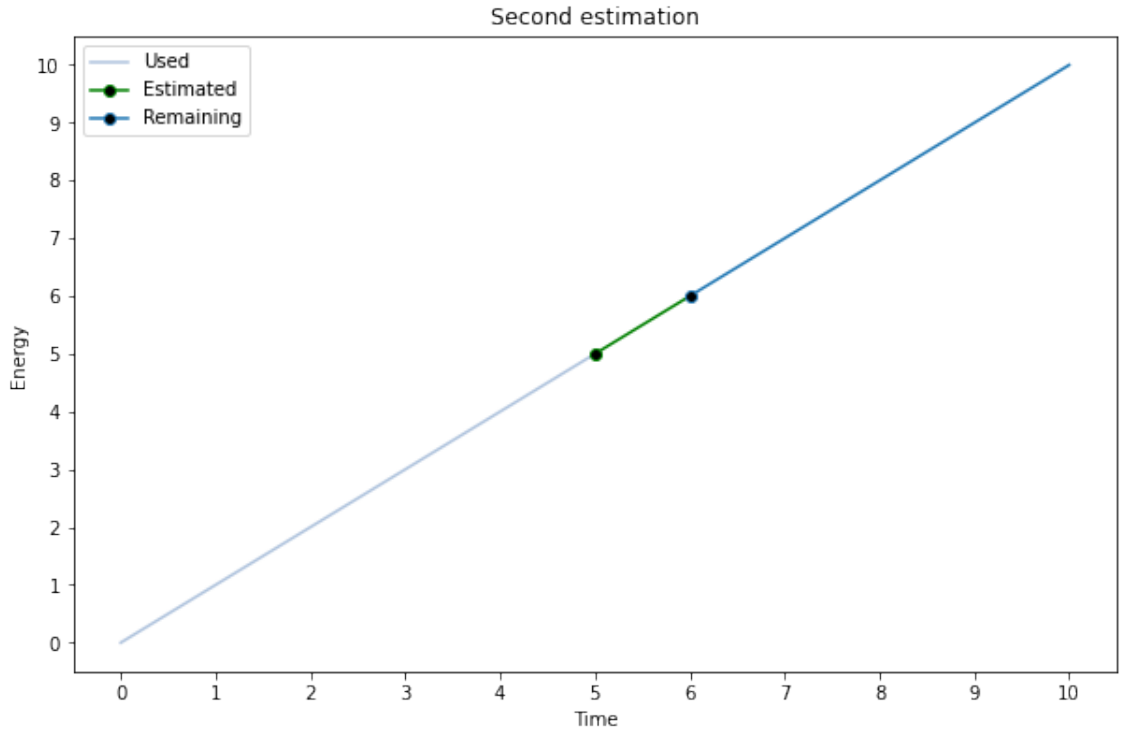


Figure 4.4: Example of second estimation

With all of the parts of Thor implemented, the implementation then has to be tested to assess if it works. The setup used for the testing is presented in Chapter 5.

Chapter 5

Experimental setup

This chapter describes the experimental setup that we use to test our implementation of Thor. This includes the system specifications for the different parts of the setup and a description of the tests we conduct.

5.1 System Specifications & Setup

The specifications of the system running Thor in the tests can be seen in Table 5.1.

Type	Desktop
CPU	Intel Core i7 11700F (2.5GHz, up to 4.9GHz)
DRAM	DDR4 32GB 2133MHz (Corsair CMK32GX4M2E3200C16)
GPU	Nvidia GeForce RTX 3060
Storage	1TB NVME SSD KINGSTON SNVS1000G
Motherboard	Asus PRIME B560M-K[1]
Operating system (OS)	Ubuntu Server 22.04.4 LTS minimal installation[2]

Table 5.1: The specification of the system-under-test

In order to test Thor, a laptop is used to run the client. The specifications of the laptop can be seen in Table 5.2.

[1]The bios was reset to default settings

[2]Information about minimal installation can be found here:
https://www.howtoforge.com/ubuntu-22-04-minimal-server/

Type	Laptop (Acer SFX14-41G-R8A1)
CPU	AMD Ryzen 7 5800U
DRAM	LPDDR4X 16GB
GPU	Nvidia GeForce RTX 3050 Ti
Storage	1TB SSD + 250GB SSD
OS	Windows 11

Table 5.2: The specification of the client

To transfer data between the client and server, a router is used. The router used for testing is a D-Link GO-RT-N300[3] which uses the 2.4 GHz band with a maximum transfer rate of 300 Mbps.

We decided to use a router to mimic a typical work environment instead of direct communication between the client and the server, because in a normal setting, there would be some form of network that the data has to traverse.

The testing is done without stopping background services. We chose not to do so, because in a production environment it may not be possible to do so. By testing Thor with the pollution from the background services and processes present, we demonstrate that Thor is able to find hotspots regardless of the noise.

As explained in Section 4.1.2 the sampler requires an interval to be set. For all tests conducted, this variable was set to 50 microseconds to capture when the MSR changed.

5.2 Tests

To assess Thor's ability to detect hotspots, we conduct a set of tests. These tests can be seen in Table 5.3.

A mix of micro-benchmarks is used to verify that Thor can find hotspots, and two implementations of a web service are used to test the usability of profiling with Thor. The implementation of the tests can be found in our repositories[4], where each repository contains a script, such that Thor can run them automatically.

[3]https://www.dlink.com/uk/en/products/go-rt-n300-wireless-n-300-easy-router
[4]https://github.com/orgs/cs-24-pt-10-01/repositories

Name	Description	Source
Micro-Benchmark mix	A mix of micro-benchmarks where the micro-benchmark which uses the most energy is known. The benchmarks are: Fibonacci, N-Body, MergeSort, and QuickSort	Rosetta Code[5] and CLBG[6]
RealWorld web service	Backend implementations of a web service in JavaScript and C# based on the RealWorld API specification.	RealWorld's GitHub repository[7][8]
Overhead test	Measuring the energy consumption of calls to Thor	Our own implementation [9]

Table 5.3: Overview of tests

5.2.1 Micro-benchmark mix

The first test is a mix of different micro-benchmarks. Using micro-benchmarks allows for a controlled environment where it can be checked if Thor can detect energy hotspots. This test will use micro-benchmarks that have previously been benchmarked for their energy consumption[28]. This means that we know what Thor should report and can therefore compare the results with what is expected. Secondly, this test will also be used to compare Thor against the method introduced by Jagroep et al.[4]. Their method requires stubbing of the code tested. The micro-benchmarks we have chosen are simple to stub and evaluate, compared to if larger programs were used. How the micro-benchmarks were stubbed is explained in Appendix A. The benchmarks in the mix are from Rosetta Code[37] and the Computer Language Benchmark Game (CLBG)[38]. The micro-benchmarks are Fibonacci, N-body, MergeSort, and QuickSort. All the micro-benchmarks require inputs. We use inputs from other work as the inputs were shown to allow the micro-benchmarks to run for a long enough period of time such that proper measurements could be gathered with RAPL[28]. The inputs can be seen in Table 5.4.

[5]https://rosettacode.org/
[6]https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html
[7]https://github.com/avanelli/fastify-realworld-example-app
[8]https://github.com/Erikvdv/realworldapiminimal
[9]https://github.com/cs-24-pt-10-01/overhead-test

Benchmark name	Input
Fibonacci	47
N-body	50,000,000
MergeSort	A list of 40,000 random integers
QuickSort	A list of 40,000 random integers

Table 5.4: Benchmarks and their associated inputs

The micro-benchmarks are written in JavaScript and are executed using Node.js version 20.12.2.

The test is split into two parts, the first part uses Thor and the second part uses the method by Jagroep et al. All micro-benchmarks are placed within their own file, which is called from a main file, such that the micro-benchmarks together create a single program. In the first part, Thor instruments every function call within the main file. In the second part of the test, the main file is executed multiple times where, for each execution, one of the benchmarks is stubbed. An additional execution where no benchmarks are stubbed is executed to gather a baseline.

A second variant of this test is executed, where MergeSort is changed to be the primary hotspot. MergeSort is chosen as it is one of the benchmarks with the lowest energy consumption. To make MergeSort the primary hotspot, a loop before the sorting is added. This loop contains 30,000 deep clones of the input. 30,000 was chosen during test executions where it was shown to be consistently adding enough overhead. This is documented in Appendix D. Using deep clones does not change the input, but it adds enough workload to make MergeSort the function with the highest energy consumption.

During testing, it was discovered that the number of recursive calls within Fibonacci was too excessive for us to handle. This is explained in Appendix B. To remedy the excessive number of calls, we added a parameter to our JavaScript instrumentation tool to indicate whether function calls within functions should be instrumented.

The test, and its variants, are repeated 10 times each to account for possible variances between executions. The number of repetitions is not important as the goal of the test is to find hotspots and not to accurately measure the energy consumption of the implementation of the benchmarks. A single execution could therefore be used, but since it is possible for outliers and random variance to occur, we chose to have multiple executions during our testing. When using Thor, a single execution is enough to find the hotspots.

Multithreading

To test Thor's ability to profile programs with multithreading, we conduct a test where the micro-benchmarks are profiled in parallel to see how parallelization affects Thor's ability to find hotspots. In this test, the main file uses worker threads from Node.js to execute each micro-benchmark at once. This test is repeated 10 times to account for possible variances.

5.2.2 RealWorld web service

The second test is to profile implementations of a web service. The implementations are part of a collection called RealWorld[10], which contains implementations of a clone of the blogging website Medium.com[11][39]. All these implementations expose the same API. The frontend and backend between them can vary and be interchanged. Instead of using a frontend, the focus is only on the backend implementation. Therefore, a client is simulated by calling the API endpoints. In this test, two implementations of the backend are profiled. The purpose of using two RealWorld implementations is to show that Thor can detect hotspots in multiple programming languages.

The first chosen RealWorld implementation is written in JavaScript and uses an SQLite database. The SQLite database does not require an external connection to a running database, which removes a variable from the test. The JavaScript implementation is instrumented using our tool explained in Section 4.2.1.

The second RealWorld implementation is written in C#, and also uses an SQLite database. This implementation is instrumented by using the AOP tool called Metalama. A look into the compile-time generated code can be seen in Appendix C.

The backend implementation is tested by using the tool called Newman[40], which is a command-line runner for the program Postman[41]. Postman allows crafting and sending requests to sites, in order to simulate a client. In the repositories of the two RealWorld implementations is a collection of requests to all endpoints in the API, these are used with Newman to send requests.

The endpoints of the RealWorld API are each called 10 times to account for possible variances and to allow for JIT compilation to take effect.

Jagroep et al.'s method[4] encounter problems when trying to use it to profile the functions of the RealWorld implementation, which is why we do not use their

[10] https://codebase.show/projects/realworld
[11] https://medium.com/

method for profiling the implementations. The reason is explained in Appendix F.

5.2.3 Overhead test

In order to determine the energy impact of using Thor, a test is performed where the energy consumption of calls to Thor is measured. This test will show the overhead introduced by Thor, which is present within the other tests.

Within the test, a pair of `start` and `stop` calls are repeated 10,000 times using a for-loop. As each call is too fast for RAPL to measure, the energy consumption of the loop is captured. To account for variances, this test is repeated 10 times.

The implementation of the overhead test can be seen in Listing 5.1.

```
1  const rapl = require('./rapl.js');
2
3  rapl.start("Main")
4
5  for (let i = 0; i < 10000; i++) {
6      rapl.start("Call");
7      rapl.stop("Call");
8  }
9
10 rapl.stop("Main")
```

Listing 5.1: Overhead test implemented in JavaScript

With the experimental setup, we conduct the tests described. The results from the tests are presented in Chapter 6.

Chapter 6

Results

With the experiemental setup described in Chapter 5, we conduct the tests to verify that Thor is able to find hotspots within a program, and we compare Thor to an existing method to ascertain the accuracy of Thor. Additionally, the results from the overhead test are presented, including the effect of estimation.

6.1 Micro-Benchmark mix

The results from the micro-benchmark mix tests can be seen in Table 6.1 and 6.2. In these tests, the expected outcome is that Fibonacci will have the highest energy consumption, and N-body will have the second highest. The expected outcome is gathered from our previous work[28].

Benchmark	**Identifier**	**Accumulated [Joules]**	**Avg. Per Call [Joules]**	**Call count**
Fibonacci	20:main.js:fib	10547.64	1054.76	10
N-body	21:main.js:nbody	1734.92	173.49	10
QuickSort	23:main.js:sort	4.18	0.42	10
MergeSort	24:main.js:mergeSortInPlaceFast	3.86	0.39	10

Table 6.1: The four most expensive function calls found from profiling the micro-benchmarks with Thor over 10 executions.

From Table 6.1 it can be seen that the expected results are achieved. This can be identified by looking at the two most expensive function calls which are Fibonacci and N-body. The identifier column shows the line of the function, the file, and the name of the function.

Name	Accumulated [Joules]	Average [Joules]	Difference [Joules]
No stubbing	12467.66	1246.77	0
MergeSort Stubbed	12386.96	1238.70	8.07
QuickSort Stubbed	12207.59	1220.76	26.01
N-body Stubbed	10661.76	1066.18	180.59
Fibonacci Stubbed	1706.86	170.69	1076.08

Table 6.2: Results from profiling with the method by Jagroep et al. over 10 executions. The Difference column is No stubbing's average minus the row's average.

The results from using the method by Jagroep et al. can be seen in Table 6.2. This method uses stubbing, so the execution with the lowest energy usage is the one with the hotspot. Alternatively, it is also the function with the highest difference. The usage of this method also achieves the expected results as the lowest energy consumption is seen when Fibonacci is stubbed and the second lowest is seen when N-body is stubbed. When comparing the results from Thor, seen in Table 6.1, and the method by Jagroep et al., seen in Table 6.2, it can be noticed that the same hotspots are identified. This shows that Thor is equally able to find hotspots to that of Jagroep et al.'s method. Additionally, when comparing the difference in Table 6.2, with the average from Table 6.1, it can be seen that the values for Fibonacci and N-Body are similar. The values for MergeSort and Quicksort are slightly higher when using Jagroep et al.'s method. The notable change when using Jagroep et al.s method, is that the difference between MergeSort and QuickSort is significant compared to the results from Thor. A possible explanation for the difference, could be caused by the variance between executions. The variance can be seen on Figure 6.1, where the difference between the maximum and minimum energy consumption is over 100 joules.

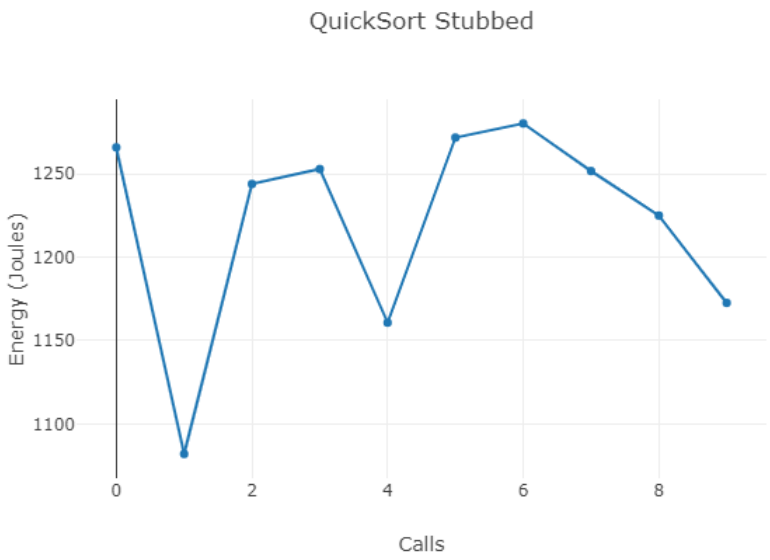


Figure 6.1: Energy consumption from the executions of the microbenchmarks with QuickSort stubbed.

The results from the second variant can be seen in Table 6.3 and 6.4 for Thor and Jagroep et al.'s method respectively. It can be seen that the additional workload added to MergeSort is caught by both, as MergeSort is identified as the primary hotspot by both methods.

Benchmark	**Identifier**	**Accumulated [Joules]**	**Avg. Per Call [Joules]**	**Call count**
MergeSort	24:main.js:mergeSortInPlaceFast	11167.58	1116.76	10
Fibonacci	20:main.js:fib	9923.66	992.37	10
N-body	21:main.js:nbody	1505.59	150.56	10
QuickSort	23:main.js:sort	3.73	0.37	10

Table 6.3: The four most expensive function calls found from profiling the micro-benchmarks with Thor over 10 executions. MergeSort contain additional workload to make it the biggest hotspot

Name	Accumulated [Joules]	Average [Joules]	Difference [Joules]
No stubbing	23767.96	2376.80	0
QuickSort Stubbed	23700.07	2370.01	6.79
N-body Stubbed	22096.30	2209.63	167.17
Fibonacci Stubbed	12838.47	1283.85	1092.95
MergeSort Stubbed	12332.99	1233.30	1143.5

Table 6.4: Results from profiling with the method by Jagroep et al. over 10 executions. The Difference column is No stubbing's average minus the row's average. MergeSort contains additional workload to make it the biggest hotspot

Multithreading test

The results from the multithreading test can be seen in Table 6.5.

From the results, it can be noticed that the energy usage of N-body has increased, and MergeSort and QuickSort have doubled their energy usage in comparison to Table 6.1. Since threads share the MSRs for reading the energy used by the machine, the benchmarks running in parallel pollute each other. This makes the reported energy usage of the function higher than it actually is. Executing the benchmarks in parallel did not change the first two hotspots, but MergeSort is indicated to have a higher energy usage than QuickSort. This indicates that parallel execution affects the profiling of functions with a low energy consumption more than functions with a higher energy consumption. Due to not being able to precisely profile all functions in a multithreaded program, using Thor on such programs requires the developer to understand the implications.

Benchmark	Identifier	Accumulated [Joules]	Avg. Per Call [Joules]	Call count
Fibonacci	9:Fib.js:fib	10438.30	1043.83	10
N-body	181:Nbody.js:N_Body	2279.02	227.90	10
MergeSort	30:MergeSort.js:mergeSortInPlaceFast	8.83	0.88	10
QuickSort	44:QuickSort.js:sort	8.29	0.83	10

Table 6.5: The four most expensive function calls found from profiling the micro-benchmarks in parallel with Thor over 10 executions.

6.2 RealWorld web service

The results from profiling the RealWorld web services can be seen in Table 6.6 and 6.7. The instrumentation of the services vary between the JavaScript and C# implementations. This results in different identifiers for the two implementations.

Identifier	Accumulated [Joules]	Avg. Per Call [Joules]	Call count
14:./index.js:listen	51.29	51.29	1
50:./lib/routes/users/index.js:hash	33.83	1.69	20
7:./lib/plugins/bcrypt/index.js:hash	33.83	1.69	20
10:./lib/plugins/bcrypt/index.js:compare	33.20	1.66	20
12:./lib/plugins/knex/index.js:run	31.24	31.24	1
2:./lib/plugins/bcrypt/index.js:require	12.19	12.19	1
18:./lib/routes/articles/index.js:getArticles	5.38	0.05	110
1:./index.js:require	2.91	2.91	1
1:./lib/config/config.js:require	2.76	2.76	1
11:./index.js:undefined	2.25	2.25	1

Table 6.6: The ten most expensive function calls of the JavaScript implementation of the RealWorld API. undefined means that the function is inline, which do not have an identifier.

From Table 6.6 it can be seen that the `listen` function has the highest energy usage within the JavaScript implementation. This function initiates the service and begins listening for requests from clients. It can be seen that hashing has a high energy consumption, as two functions used for hashing shares the second highest accumulated consumption. This is to be expected, as the first function, which is from `users`, results in a call to the other `hash` function of the `bcrypt` plugin. The function with the fourth highest accumulated energy usage is the `compare` function from the `bcrypt` plugin, this function compares plaintext with hashed information. The fifth highest accumulated usage comes from the `run` function of the `knex` plugin, which is used to start the SQLite database. From these hotspots it can be seen that the JavaScript implementation uses the most energy on the initial launch and encryption. Since it is a service, the longer the service is running the less the initial launch will have an effect on the overall energy consumption.

The results from the C# RealWorld example can be seen in Table 6.7.

Identifier	Accumulated [Joules]	Avg. Per Call [Joules]	Call count
Main	403.44	403.44	1
UserHandler.CreateAsync	13.35	0.67	20
ArticlesHandler.GetArticlesAsync	11.68	0.10	120
ArticlesHandler.CreateArticleAsync	4.89	0.49	10
UserHandler.LoginAsync	2.90	0.14	20
ArticlesHandler.AddCommentAsync	1.73	0.17	10
ArticlesHandler.GetArticleBySlugAsync	1.59	0.16	10
ArticlesHandler.AddFavoriteAsync	1.05	0.10	10
ArticlesHandler.DeleteFavorite	0.91	0.09	10
ArticlesHandler.DeleteArticleAsync	0.84	0.08	10

Table 6.7: The ten most expensive function calls of the C# implementation of the RealWorld API

The most energy consuming function is `Main`. The `Main` function executes the whole logic of the program. `Main` can be disregarded as a hotspot, due to it not containing any computationally heavy code of its own. Disregarding `Main` requires the developer to have sufficient knowledge of the source code, which is an aspect of Thor that is discussed in Section 7.5. What is noticeable is that despite certain methods, such as the `CreateAsync` method of the `UserHandler` class as seen in Figure 6.2, using the most energy, it has a sharp decline after the initial call. The initial call consumes 12.26 joules, where subtracting the accumulated energy consumption from the initial call results in an average consumed joules per call of 0.06.

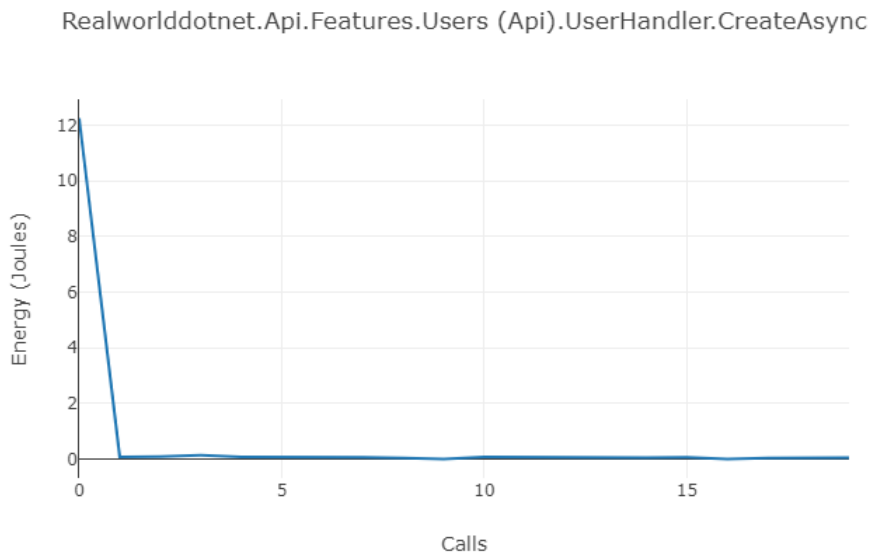


Figure 6.2: Graph of the energy consumption of CreateAsync calls.

Another hotspot based on accumulated energy consumption, which is the `GetArticlesAsync` method of the `ArticlesHandler` class, as seen in Figure 6.3, also has a sharp decline in its energy consumption after the few initial executions.

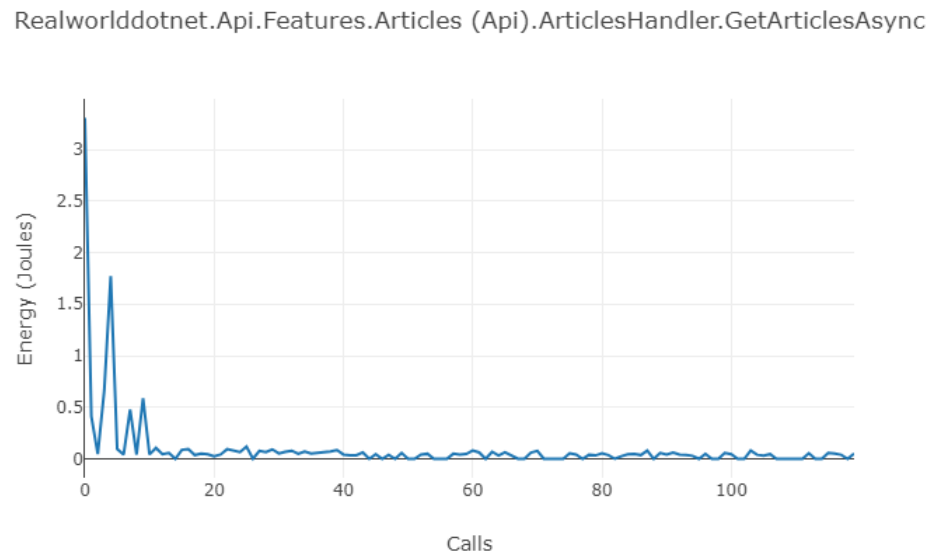


Figure 6.3: Graph of the energy consumption of GetArticlesAsync calls.

Lastly, the `CreateArticleAsync` method of the `ArticlesHandler` class, as seen in Figure 6.4, follows the same pattern as the `CreateAsync` method. There is a large upfront cost to the method, but after the initial call, its energy consumption per call decreases. The initial call consumed 4.04 joules, where subtracting the accumulated energy consumption from the initial call results in an average consumed joules per call of 0.09. The decrease in energy consumption seen with the mentioned methods can be attributed to initialization or JIT compilation.

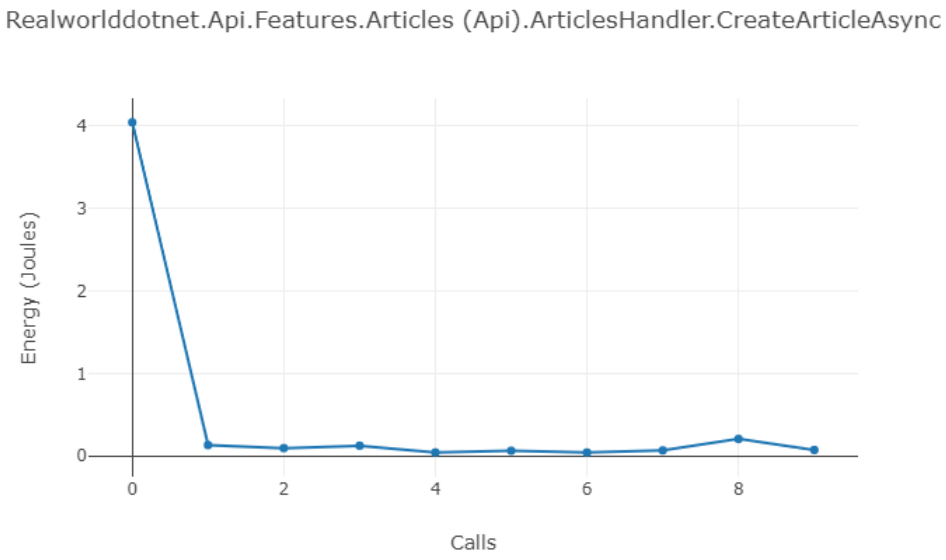


Figure 6.4: Graph of the energy consumption of CreateArticleAsync calls.

From the results of profiling the two RealWorld API implementations, it can be seen that Thor can find functions with high energy usage. These functions are the hotspots in the implementations.

6.3 Overhead test

The results of the overhead test can be seen in Figure 6.5. The accumulated energy of the calls outside the loop, indicated by the `main` identifier, which can be seen in Listing 5.1, is 13.88 joules, which means the average over the 10 executions is 1.388 joules. Since there are 10,000 pairs of start and stop calls, each pair, on average, consumes 138.8 microjoules. The average energy consumption reported by the inner function calls, indicated by the `call` identifier, is 71.9 microjoules. Since the MSR is accumulating, it is unclear why there is a noticeable difference between the outer and inner calls. In comparison to the results shown in the rest of this chapter,

the average consumption of start and stop calls is too small to have a noticeable effect. This suggests that the overhead introduced by Thor had an insignificant effect on the other results.

From Figure 6.5 it can be seen that the energy consumption of the 10,000 pairs varies. The maximum consumption of an execution is 1.505 joules and the minimum is 1.173 joules. The difference between maximum and minimum is 0.332 joules, because these executions contains 10,000 pairs of calls it means that there are, on average, a 33.2 microjoules difference between the pairs of calls.

While the variation can affect hotspot detection for calls with a difference within 33.2 microjoules, any hotspots with an energy consumption that close to each other, can be seen as equivalent.

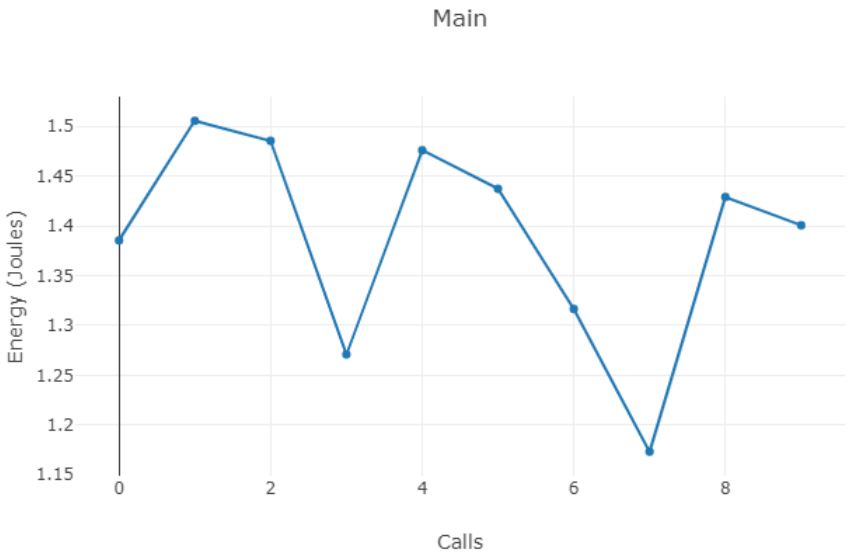


Figure 6.5: Graph of the energy consumption of the overhead test executions.

6.4 Effect of estimation

This section shows the effect of the estimation functionality explained in Section 4.5.

The overhead test contains 10,000 measurements which are too fast for RAPL to measure, but they can be estimated. Figure 6.6 shows the measurements without estimation, where it can be seen that the change in energy consumption is not always captured. The estimation of these can be seen in Figure 6.7. The average of

the estimated energy consumption is 69.5 microjoules, which is close to the average of the inner calls presented in the overhead test.

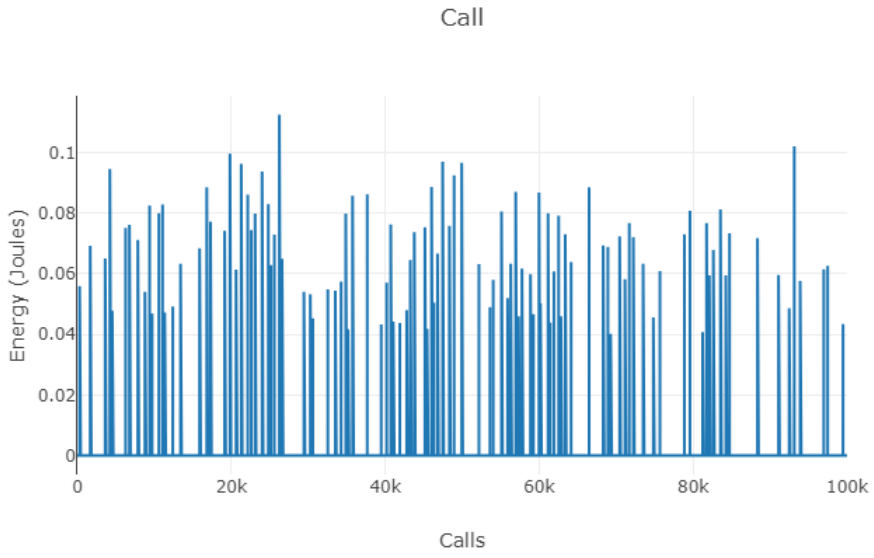


Figure 6.6: Measurements from the overhead tests calls without estimation.

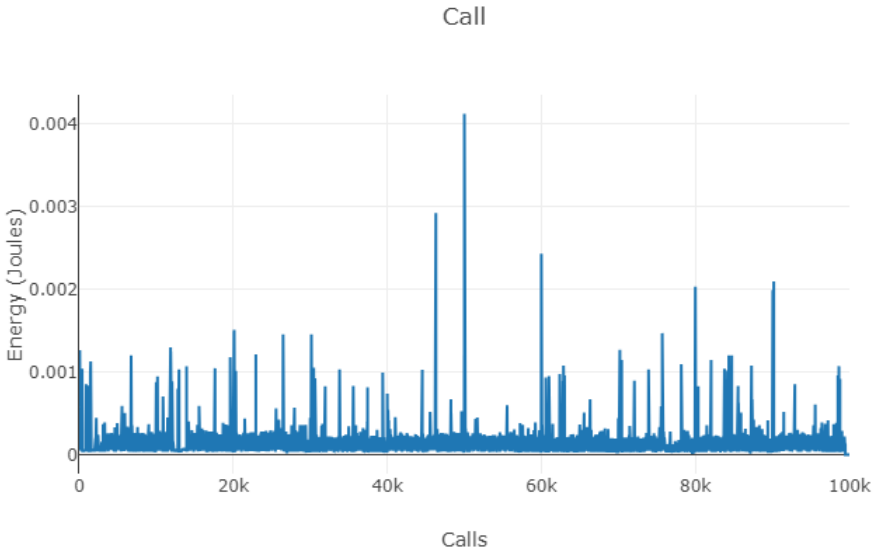


Figure 6.7: Estimation of the energy consumed for each call in the overhead test.

To see if estimation has an effect on the identified hotspots, we present the hotspots

from the RealWorld implementations with and without estimation. These can be seen in Table 6.8 and 6.9.

Estimated		**Not Estimated**	
Identifier	**Accumulated [Joules]**	**Identifier**	**Accumulated [Joules]**
14:./index.js:listen	51.29	14:./index.js:listen	51.29
50:./lib/routes/users/index.js:hash	33.56	50:./lib/routes/users/index.js:hash	33.83
7:./lib/plugins/bcrypt/index.js:hash	33.56	7:./lib/plugins/bcrypt/index.js:hash	33.83
10:./lib/plugins/bcrypt/index.js:compare	32.98	10:./lib/plugins/bcrypt/index.js:compare	33.20
12:./lib/plugins/knex/index.js:run	31.24	12:./lib/plugins/knex/index.js:run	31.24
2:./lib/plugins/bcrypt/index.js:require	12.18	2:./lib/plugins/bcrypt/index.js:require	12.19
18:./lib/routes/articles/index.js:getArticles	5.98	18:./lib/routes/articles/index.js:getArticles	5.38
72:./lib/models/articles.js:first	**2.93**	1:./index.js:require	2.91
1:./index.js:require	2.92	1:./lib/config/config.js:require	2.76
1:./lib/config/config.js:require	2.76	11:./index.js:undefined	2.25

Table 6.8: The ten most expensive function calls of the JavaScript implementation, with and without estimation. Significant changes are highlighted with bold text.

From Table 6.8 it can be seen that estimation introduces a new function to the ten most expensive functions of the JavaScript implementation. This is the `first` function from `articles.js`. Without estimation, this function is measured to 2.19 joules and thereby not one of the ten most expensive, but it is estimated to consume 2.93 joules. Some existing hotspots are also estimated to have a higher accumulated energy consumption. These are the two hash functions, the `compare` function, and the `getArticles` function.

Estimated		**Not Estimated**	
Identifier	**Accumulated [Joules]**	**Identifier**	**Accumulated [Joules]**
Main	403.44	Main	403.44
UserHandler.CreateAsync	13.42	UserHandler.CreateAsync	13.35
ArticlesHandler.GetArticlesAsync	12.56	ArticlesHandler.GetArticlesAsync	11.68
ArticlesHandler.CreateArticleAsync	4.89	ArticlesHandler.CreateArticleAsync	4.89
UserHandler.LoginAsync	3.07	UserHandler.LoginAsync	2.90
ArticlesHandler.AddCommentAsync	1.73	ArticlesHandler.AddCommentAsync	1.73
ArticlesHandler.GetArticleBySlugAsync	1.66	ArticlesHandler.GetArticleBySlugAsync	1.59
ArticlesHandler.AddFavoriteAsync	1.05	ArticlesHandler.AddFavoriteAsync	1.05
ArticlesHandler.GetCommentsAsync	**0.92**	ArticlesHandler.DeleteFavorite	0.91
ArticlesHandler.DeleteFavorite	0.91	ArticlesHandler.DeleteArticleAsync	0.84

Table 6.9: The ten most expensive function calls of the C# implementation, with and without estimation. Significant changes are highlighted with bold text.

Similarly to the JavaScript implementation, estimation introduces a new function to the ten most expensive functions in the C# implementation, the results of which can be seen in Table 6.9. The method that was added is the `GetCommentAsync` method from the `ArticlesHandler` class. This function is measured to 0.81 joules which is not within the ten most expensive functions, but it is estimated to use

0.92 joules which is within the ten most expensive functions. An existing hotspot that is estimated to have a higher consumption, but without a change in position, is the `GetArticlesAsync` method from `ArticlesHandler` class, which is measured at 11.68 joules and estimated to be 12.56 joules.

From the presented estimations, it can be seen that the estimation functionality can be used to estimate the energy consumption of fast functions, to an extent that can highlight different hotspots.

Chapter 7

Discussion

This chapter first presents limitations of Thor in regards to a production environment. Then, some of the important aspects of Thor are discussed. Finally, threats to validity are presented.

7.1 Limitations of Thor in a production environment

This section presents and discusses parts which are either needed or nice to have in a production environment. These parts are not necessary to make Thor work but they provide various benefits.

7.1.1 Instrumentation tools

To profile with Thor, the program to be profiled has to be instrumented. This means that the profiling capabilities of Thor are limited to the instrumentation within the process-under-test. Therefore the capability of Thor is limited by the instrumentation tools that are available. Manual instrumentation is possible and extends the profiling capability of Thor but would require that the developer spends their time instrumenting. For small programs this might not be a problem, but for larger programs, depending on how much of the program the developer wants to instrument, it could require a substantial amount of development time.

Since Thor can be extended with additional instrumentation tools, existing tools could be added or new tools could be created. These tools could focus on instrumenting the specific parts the developer wants to instrument. It could also include support for additional programming languages.

7.1.2 Security

Measurements of energy consumption can be used as side channel attacks[42]. The implications of this limits the usability of Thor. The way Thor's networking is designed, it can be used over both open and closed networks. If the server is exposed to the web and allows users to access energy measurements, then it allows for side channel attacks. The server should only be accessible to a closed network to stop attackers from accessing energy measurements. Another security concern is the execution of user code. An attacker can send repositories with malicious code to the server to be executed. This is another reason for not exposing the server unprotected to the web.

A security layer requiring authentication and authorization before access could remedy this. This is not something that is currently implemented in Thor, but for use in a production environment, such a security layer should be implemented. Adding a security layer, would require an extension to the `Listener` component, specifically to the connection to the client, but the rest of Thor does not require change. Ways to implement such a security layer are presented in Section 8.1.

7.1.3 Containerization & Cloud

In a production environment, developers might want to use Thor within an isolated environment using virtualization or containerization[43, 44].

An issue with these technologies is that the MSRs are not always available. It is possible to enable access to the MSRs in an isolated environment, but it requires disabling security features. An additional concern is that while a host can run multiple isolated environments, if an isolated environment attempts to acquire the energy measurement, the energy consumption will account for all running environments. This is further explained in Section 7.1.4.

Cloud providers, such as Amazon Web Services, do not provide access to Intel's RAPL, or similar interfaces, due to security concerns[45]. This means that it is important when performing energy measurements to have access to the machine the software is running on, as a cloud provider can prevent access to the necessary interfaces. If a company does not have access to the machine their software is running on, they could use Thor on a different system. This does come with the risk of the results not representing how the software will run in a production environment. If the system is similar to the machine used in production then the results can be expected to be similar. If hardware designed for specific purposes is used, such as hardware for cryptography, then these should be included to mitigate potential disparity.

It is possible for an isolated environment to send requests to the host with TCP

sockets. This communication can, for example, be `start` and `stop` packets to Thor, however, it does come with the cost of additional overhead[46]. The size of the overhead is unknown and would require tests to assess. Depending on how big the overhead is, it may or may not affect the ability of Thor to identify hotspots. For the overhead to have an effect, it has to be big enough to pollute the measurements to a point where distinctive hotspots cannot be found. This is primarily an issue if the program consists of many functions with a low energy consumption, which are executed many times.

7.1.4 Profilling more than one program

In the executed tests, only one program is executed at a time. The method of acquiring energy measurements occurs through the MSR for the package energy.

This MSR accounts for the energy usage as a whole. All processes or services running will cause this MSR's value to increase, which include more than the process executing the test. This pollutes the measurement.

A way to filter out this pollution, along with adding the possibility of testing multiple programs at the same time, could be to analyze the CPU utilization of processes and services. This is because the energy consumption depends on the utilization of the processes and services on the CPU. A possible solution to this problem is presented in Section 8.1.

7.2 Resource overhead

When profiling with Thor, the process-under-test has to send `start` and `stop` calls to the server, which introduces an overhead in resource usage. In some cases where functions are called many times, the overhead introduced might make the profiling unfeasible. In the case with Fibonacci, which is presented in Appendix B, we encountered problems with resource usage.

The amount of calls can be reduced in multiple ways. One way would be to add a filter to the server, such that fast function calls can be filtered out. An issue with this approach is that the accumulated energy usage of these fast calls can be enough to be a hotspot.

Another way to reduce the amount of calls is to instrument fewer function calls. This was the approach taken to make Fibonacci feasible, where only the outer call to Fibonacci was instrumented and not the recursive calls.

Alternative ways of instrumenting code are presented in Section 8.1.

7.3 Async functions

Async functions present an interesting difficulty for instrumentation. If an async function that returns a promise is instrumented, then it will only result in measurements for the consumption of returning a promise. This means that it is important to instrument how the promise is resolved in order to obtain the energy consumed.

An issue that can occur when profiling async functions with Thor is that the same thread can make multiple start calls for the same function before reaching a stop. This results in an issue where we cannot determine which stop call is for which start call. A possible solution is presented in Section 8.1.

7.4 MSR update interval

As mentioned in Section 4.4, the update interval for the MSR, is not fast enough to capture all functions, resulting in wrongful results of zeros. This happens when both the start and stop of a function happens within a single update of the MSR.

We added an optional functionality to estimate the energy consumed between measurements. This is detailed more in Section 4.5. This functionality allows for estimation of the energy consumption of functions which would otherwise be measured to zero.

Without the estimation, Thor is still able to fulfil its purpose of finding hotspots. The estimation does have a small impact which is explained in Section 6.4. The estimation did not have an impact on the ranking of most energy consuming functions, but for the lower ranked functions, a change in ranking occurred.

For general measurement of the energy consumption of software, this feature can provide a way to look into energy consumption of fast functions without the need for repetitions. Future work can look into how this feature can be used to capture energy consumption of fast functions. A comparison between repetitions and estimation can provide insights into the accuracy of estimating the energy consumption of fast functions.

7.5 Knowledge of the code

The resulting hotspots reported by Thor might not always be accurate. As seen in Section 6.2, the `Main` function was shown as the biggest hotspot. Since the `Main` function consisted mostly of other function calls, the reported energy consumption

includes the consumption of those function calls. For a developer to locate expensive parts of the code from the results, they will have to understand the code of the parts that are reported as hotspots. For code that the developer has written themselves this should not be a problem, but if it is written by a team of developers, there might be parts where the individual developer will have to analyze and understand the code before it is certain that it is a hotspot.

Handling nested function calls automatically

It is possible to automatically determine the energy usage of a function call without the inner calls. It would require what we call a function-trace, for each function to be known. A function-trace, is a list of function identifiers, where each subsequent function is nested within the previous function. If the function-trace is known then the energy consumption from the function calls can be subtracted from the function's energy consumption, resulting in just the function's own energy consumption. Adding a function-trace would therefore be beneficial as it allows functions, such as `Main`, to show their own energy consumption. Possible ways to generate a function-trace are presented in Section 8.1.

Still, from our results, if the developers understand the code that is tested, this is not an issue, but it would allow for further automation.

7.6 Threats to validity

This section will present and discuss threats to the validity of our results.

7.6.1 Internal Validity

There are aspects within the internal setup of our study that can present threats to validity.

Validity of results

If the measurements are not accurate enough, it would threaten the validity of the results and the ability of Thor to find hotspots. As mentioned in Section 4.1.2 Intel's RAPL has been tested to be as accurate as measurements from a power plug[27], which is enough to detect hotspots.

In Section 5.1 it is explained that the sampler is set to take a measurement every 50 microseconds. It can be argued that a sample rate under 1 kHz yields redundant measurements and introduces a larger overhead. However, a high sample rate ensures that the calls from the process-under-test are matched with a newer measurement if the MSR is updated just before the call.

Too much noise, or pollution, in the measurements can threaten the validity of the results. This leads to the question of how much noise is in the measurements, as both Thor and the OS consumes energy under testing. In the overhead test, the overhead of multiple calls to Thor was measured. The results of the test seen in Section 6.3 shows an average consumption per call that is too small to have a noticeable effect on the results.

In the multithreading test, it was found that parallel executions did introduce noise, as all micro-benchmarks, except Fibonacci, yielded a higher energy consumption. In the case of this project, noise is acceptable as long as it does not hinder hotspot detection.

The results from Thor in the micro-benchmark mix are validated by producing results in correlation with both another method and measurements of the same benchmarks from another study.

Validity of hotspots identified by Thor

The validity of the identified hotspots in the RealWorld implementations is based on the ability of Thor to identify hotspots. As the micro-benchmark mix verifies that Thor can identify hotspots, the validity of hotspots identified in the RealWorld implementations is dependent on the results of the micro-benchmark mix. From the results of the micro-benchmark mix, it was verified that Thor can detect hotspots. Because of this, the validity of the results from the RealWorld examples has been shown.

The results from the multithreading test shown in Section 6.1, illustrated how parallel execution can affect Thor's ability to find hotspots. This introduces a threat to validity regarding hotspots with lower energy consumption as MergeSort and QuickSort were affected. This does affect the results seen from the RealWorld implementations tested, as both utilize multiple threads. The size of this effect is small, as the difference between the average energy consumption of MergeSort and QuickSort was 0.03 joules in the micro-benchmark test and 0.05 joules in the multithreaded test. These results indicate that the ranking of larger hotspots is not noticeably affected.

The results from the C# implementation of the RealWorld API presents a challenge. As explained in Section 6.2, the hotspots that were identified has the highest energy consumption during the first calls, whereas the later calls uses less energy. This presents the question of whether other hotspots would be found if the endpoints were called more times. Future work can look into how variables, such as the amount of times an endpoint is called, affect the identified hotspots within the tests.

Structured testing of Thor's implementation

During the development of Thor we used manual testing to assess if our implementation worked correctly. Because of this, there could be parts or specific scenarios that we missed. Using automated tests could, for example in a Continuous Integration (CI) pipeline, increase the likelihood that errors would get caught, provided that appropriate tests were set up. Due to Thor requiring access to the MSRs, setting up CI, or automated tests in general, becomes problematic if the tests are not executed on a system where access to the MSRs is granted. This is further explored in Section 8.1.

From the tests we conducted, we encountered no issues with Thor, indicating that our manual testing was sufficient.

Using the filesystem for retrieving RAPL measurements

As mentioned in Section 4.1.2, there are multiple ways to read the RAPL measurements. The experimental setup of Thor uses the Ubuntu server OS. The implementation of Thor reads the RAPL measurements from a file. This means that the filesystem has to be accessed while reading measurements. Despite the system using a sampler, the usage of the filesystem could introduce a bottleneck to the speed at which the measurements can be read. Accessing RAPL measurements using the kernel command `rdmsr` requires using a driver. The driver could be developed to contain the sampler. This would remove the need for the sampler to call the driver which would further reduce the potential bottleneck.

From the executed tests, it was found that reading the measurements from the filesystem did not cause any issues. As such, developing a driver was not warranted, as that would take time away from developing the implementation of Thor.

7.6.2 External Validity

There are aspects of the experimental setup, which, if changed, could change the results of the tests. If Thor was tested on a different system, it might change the results, as other systems can be less or more efficient with certain operations[47].

Thor is implemented with Intel's RAPL, which has been shown to be reliable for testing CPUs[27]. Therefore, for this project where we only test the system's CPU, RAPL is a good choice.

In the tests, programs implemented with two programming languages have been profiled with Thor. The languages we chose to use are some of the more popular

languages[48]. Choosing a popular language shows that Thor works in a similar context to what others may have.

How to generalize our findings is further detailed in Section 8.1.

Chapter 8

Conclusion & Future Work

In this project, we developed a tool named Thor, which can aid developers with finding energy hotspots in their code.

We set out to answer the following problem statement.

How can a tool, designed to find energy hotspots, be constructed to help developers gain an insight into a program's energy usage?

From this, we formulated the following two research questions.

- *RQ1: How can hotspots reliably be detected in a program?*
- *RQ2: How does the hotspot detection compare to similar tools?*

RQ1 is answered by a combination of our implementation and results. It was shown that Thor is reliable in detecting energy hotspots. The implementation uses Intel's RAPL to get the energy consumption. This means that the implementation used is able to find hotspots without the use of energy estimation.

RQ2 is answered by conducting the tests explained in Section 5.2.1. The results from the tests are presented in Section 6.1. Here it can be seen that the results produced by Thor and the results produced by Jagroep et al.'s[4] method are similar. The only noticeable difference is that Thor reports lower energy consumption for QuickSort and MergeSort, but they are still in the same order.

Having developed a tool to answer these research questions, we have answered the problem statement.

The main contribution of the project is Thor, which is able to find energy hotspots within programs written in either C# or JavaScript and can be extended to support more programming languages. To support a new programming language a FFI

has to be made for the language in the case that the language does not support it, and a new instrumentation tool has to be included.

An additional contribution is a client for Thor, which allow developers to view the results in real-time. The client is in the form of a Visual Studio Code extension. The client includes the estimation functionality. This functionality was not required to answer the problem statement, but it sought to resolve an issue which could impact the results of Thor. The estimation feature did not make a significant difference in the results of our testing, but the capability still has its potential.

Comparing Thor to Jagroep et al.'s method, Thor is equally accurate in hotspot detection. Thor has the advantage that it does not require manual changing of the program and only requires a single execution, whereas Jagroep et al.'s method requires manual stubbing and multiple executions. While untested, Jagroep et al.'s method would, in theory, be able to profile multithreaded programs without any issues due to the fact that the method uses the total energy consumption of the entire execution, whereas Thor uses the energy consumption for each function. In Appendix F, it is explained how Jagroep et al.'s method can have problems with stubbing functions, but not larger components. Thor does not have a problem with measuring individual functions, and it is therefore able to measure at a finer granularity.

8.1 Future work

This section presents some of the future work presented in Chapter 7. Only the future work we considered the most interesting are presented here.

Accuracy of the energy measurements

As discussed in Section 7.6.1 in the case that a function is executed on two or more threads in parallel, issues with the measurements can occur, as is seen with the multithreading test in Section 6.1. Since all threads share MSRs for energy measurements, if both functions that are executing in parallel finishes at the same time, reading an energy measurement at the start and end of just one of the functions will show twice the energy usage.

While this energy measurement is correct in terms of the energy consumption for the CPU, it raises the question of how to more accurately measure the energy consumption on a per-thread basis. Approaches could be to look at the amount of executed instructions, or the time that a thread has been running. Tools which are able to differentiate energy usage between processes already exist. Analyzing how one of these tools, such as Scaphandre[49], differentiate between the processes, would be a good starting point. While processes and threads are not the same, it

could help create an understanding of how to differentiate energy usage based on multiple occurring events. A way to measure energy usage based on threads would give a deeper insight into energy consumption of software than Scaphandre.

This would also allow for the profiling of more than just one program at a time, which would solve the issue discussed in Section 7.1.4.

Changing instrumentation to handle asynchronous functions

A possible solution to the problem with asynchronous programming mentioned in Section 7.3 is to provide unique data to the identifier of the `start` and `stop` calls on the stack of the function. This would allow any thread which suspends or resumes the task to provide the unique details about the function. For example, one thread could report its timestamp, clock cycles, or some other metrics, while another thread that then resumes the task could also report these metrics. By using the unique data in the stack of the function, it is possible to know when the asynchronous function starts and ends, as it is no longer tied around the lifetime of the thread.

Generalizing our findings

As mentioned in Section 7.6.2 further work can be done to generalize our findings. By testing with additional applications and setups, the confidence in Thor's ability to find hotspots increases.

The additional setups can include other measurement interfaces such as Nvidia SMI[9]. Other measurement interfaces would require extending the implementation of Thor, which includes changing the implementation of the `Measurement` component.

Additional programs can include database management systems, machine learning, and more. Some of the interesting aspects that can be tested with additional programs are additional programming languages and paradigms. For example, the functional paradigm can have a high usage of recursion, which can introduce the problem with excessive recursion explained in Appendix B.

Adding a function-trace

As mentioned in Section 7.5, being able to account for nested functions would be beneficial. It would give the developer a view into the consumption of a function without the consumption of the nested functions.

This would, as was also mentioned, require function-traces. Generating function-traces can be done in multiple ways.

Knowing the function-trace at compile-time could be done by analyzing the call graph for the program. A call-graph generator, such as NDepend[50], will have to be used to create the call-graph. The runtime call-graph generators could potentially be extended to allow for interfacing during the runtime. How feasible it is to interface with a runtime call-graph generator or if the process introduces any other issues is unknown.

A stack-trace could also be used, and can be generated at runtime, which means that the tools used for instrumentation is less important. A stack trace contains more data than what is needed to construct the function-trace. This data can be filtered such that only the necessary parts are kept.

The best approach is unknown and a comparison of the two would have to be performed for a definitive answer.

Testing Thor using mocking

As mentioned in Section 7.6.1, Thor is manually tested. Adding automated tests would increase the confidence in Thor's implementation. If CI were to be implemented to enforce these tests, the confidence would increase further. Since Thor is hosted on GitHub, GitHub's CI setup could be used. If GitHub's setup is to be used, the CI runner will have to be a self-hosted runner[51] due to the lack of access to the MSRs on GitHub's runners.

It is also possible to mock the MSR, pretending that the values are accumulating, or otherwise set. This would also allow testing Thor in a CI task.

A question is then how these mocked values should be represented. They could be gathered from logging measurements over a certain period of time with varying workloads, automatically generated using, for example, randomized values, or manually crafted. It is unknown which representation would be best suited for the task.

Instrumentation tools

Additional instrumentation tools can be implemented with other languages to account for the limitation of available tools mentioned in Section 7.1.1. This project has not tested Thor with ahead-of-time compiled languages such as C and C++. These programming languages might present difficulties or new possibilities for profiling. The GCC compiler has tools to instrument code during compilation[16]. These tools can be used to profile ahead-of-time compiled programs with Thor. Another programming language where an instrumentation tool can be implemented, is Java. Instrumentation of Java will allow for comparison between Thor and Java based tools, such as eprof[10] and E-surgeon[11].

Other approaches to instrumentation can be explored as mentioned in Section 7.2. These approaches can include ways to select which functions to instrument, as instrumenting every function can become excessive as seen with Fibonacci. The addition of a maximum depth variable could be used to decide which function calls to instrument, since the depth of a function call can be calculated using a call graph. Inspiration could also be obtained from the work by Lehr et al.[15]. If functionality to analyze information gain from nested functions were to be implemented, Thor could potentially be able to automatically decide if any information is gained by going further into nested functions. This could potentially solve the problem with Fibonacci automatically, as Thor would limit itself to just the outer layer, as the nested layers do not change.

Security layer

As mentioned in Section 7.1.2 an additional security layer can be implemented into Thor.

A question to be answered is how to validate the users that should be getting the energy measurements, from those that should not.

One approach is to perform token-based user authentication with protocols such as OAuth 2.0[52]. A user can perform authentication with, for example, a username and password, where Thor could then return a token. This token can then be used to gain access to profiling.

Another approach is to take inspiration from SSH key authentication by using public and private keys[53]. The developer can then use their private key to show that they have access to use Thor.

To protect from man in the middle attacks[54], the security layer must also encrypt the data exchanged. This can be done with protocols such as the Transport Layer Security protocol[55].

Bibliography

[1] *GitHub Copilot*. URL: https://marketplace.visualstudio.com/items?itemName=GitHub.copilot.

[2] Steffen Lange, Johanna Pohl, and Tilman Santarius. "Digitalization and energy consumption. Does ICT reduce energy demand?" In: *Ecological Economics* 176 (2020), p. 106760. ISSN: 0921-8009. DOI: https://doi.org/10.1016/j.ecolecon.2020.106760. URL: https://www.sciencedirect.com/science/article/pii/S0921800919320622.

[3] Erol Gelenbe and Yves Caseau. "The Impact of Information Technology on Energy Consumption and Carbon Emissions". In: *Ubiquity* 2015.June (June 2015). DOI: 10.1145/2755977. URL: https://doi.org/10.1145/2755977.

[4] 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. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S0950584917303841.

[5] Hirst J.M. et al. "Watts Up? Pro AC Power Meter for Automated Energy Recording." In: *Behavior Analysis in Practice* 6 (1 June 2017), pp. 82–95. ISSN: 2196-8934. DOI: 10.1007/BF03391795. URL: https://doi.org/10.1007/BF03391795.

[6] Microsoft. *Joulemeter: Computational Energy Measurement and Optimization*. https://www.microsoft.com/en-us/research/project/joulemeter-computational-energy-measurement-and-optimization/. 2010. (Visited on 02/22/2024).

[7] Saurabhsingh Rajput et al. *Enhancing Energy-Awareness in Deep Learning through Fine-Grained Energy Measurement*. 2024. arXiv: 2308.12264 [cs.LG]. (Visited on 02/28/2024).

[8] Howard David et al. "RAPL: Memory Power Estimation and Capping". In: ISLPED '10. Austin, Texas, USA: Association for Computing Machinery, 2010, 189–194. ISBN: 9781450301466. DOI: 10.1145/1840845.1840883. URL: https://doi.org/10.1145/1840845.1840883.

[9] Nvidia. *System Management Interface SMI*. https://developer.nvidia.com/nvidia-system-managementinterface. 2012. (Visited on 02/29/2024).

[10] Simon Schubert et al. "Profiling Software for Energy Consumption". In: *2012 IEEE International Conference on Green Computing and Communications*. 2012, pp. 515–522. DOI: 10.1109/GreenCom.2012.86.

[11] Adel Noureddine et al. "Runtime monitoring of software energy hotspots". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE '12. Essen, Germany: Association for Computing Machinery, 2012, 160–169. ISBN: 9781450312042. DOI: 10.1145/2351676.2351699. URL: https://doi.org/10.1145/2351676.2351699.

[12] Mark Stephenson et al. "Flexible software profiling of GPU architectures". In: *SIGARCH Comput. Archit. News* 43.3S (June 2015), 185–197. ISSN: 0163-5964. DOI: 10.1145/2872887.2750375. URL: https://doi.org/10.1145/2872887.2750375.

[13] Ritu Arora et al. "Profiler instrumentation using metaprogramming techniques". In: *Proceedings of the 46th Annual Southeast Regional Conference on XX*. ACM-SE 46. Auburn, Alabama: Association for Computing Machinery, 2008, 429–434. ISBN: 9781605581057. DOI: 10.1145/1593105.1593218. URL: https://doi.org/10.1145/1593105.1593218.

[14] Thomas Ilsche et al. "Combining Instrumentation and Sampling for Trace-Based Application Performance Analysis". In: *Tools for High Performance Computing 2014*. Ed. by Christoph Niethammer et al. Cham: Springer International Publishing, 2015, pp. 123–136. ISBN: 978-3-319-16012-2.

[15] Jan-Patrick Lehr, Alexander Hück, and Christian Bischof. "PIRA: performance instrumentation refinement automation". In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*. AI-SEPS 2018. Boston, MA, USA: Association for Computing Machinery, 2018, 1–10. ISBN: 9781450360678. DOI: 10.1145/3281070.3281071. URL: https://doi.org/10.1145/3281070.3281071.

[16] *Program Instrumentation Option*. URL: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html.

[17] Git Community. *Git*. https://git-scm.com/. 2024. (Visited on 03/20/2024).

[18] Microsoft. *Instrumentation in Visual Studio (C#, Visual Basic, C++, F#)*. https://learn.microsoft.com/en-us/visualstudio/profiling/instrumentation-overview?view=vs-2022. 2024. (Visited on 03/06/2024).

[19] Oracle. *Package java lang instrument*. https://docs.oracle.com/en/java/javase/11/docs/api/java.instrument/java/lang/instrument/package-summary.html. 2023. (Visited on 03/06/2024).

[20] Gregor Kiczales et al. "Aspect-oriented programming". In: *ECOOP'97 — Object-Oriented Programming*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242. ISBN: 978-3-540-69127-3.

[21] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*. URL: https://cdrdv2.intel.com/v1/dl/getContent/671427.

[22] Vince Weaver. *Linux support for Power Measurement Interfaces*. https://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html. 2018.

[23] Rui Pereira et al. "Ranking programming languages by energy efficiency". In: *Science of Computer Programming* 205 (2021), p. 102609. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102609. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[24] E.W. Dijkstra. "Co-operating sequential processes". English. In: *Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys*. United States: Academic Press Inc., 1968, pp. 43–112. ISBN: 0-12-279750-7.

[25] Nicolás Jorge Dato. *IPC Performance Comparison: Anonymous Pipes, Named Pipes, Unix Sockets, and TCP Sockets*. https://www.baeldung.com/linux/ipc-performance-comparison. (Visited on 02/28/2024).

[26] Shaneel Narayan and Yhi Shi. "TCP/UDP network performance analysis of windows operating systems with IPv4 and IPv6". In: *2010 2nd International Conference on Signal Processing Systems*. Vol. 2. 2010, pp. V2–219–V2–222. DOI: 10.1109/ICSPS.2010.5555285.

[27] Kashif Nizam Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (Mar. 2018). ISSN: 2376-3639. DOI: 10.1145/3177754. URL: https://doi.org/10.1145/3177754.

[28] Jakob Zacho Søndergaard et al. *Measuring energy efficiency of JIT compiled programming languages with micro-benchmarks: A study into how JIT compilation affects energy consumption*. https://kbdk-aub.primo.exlibrisgroup.com/permalink/45KBDK_AUB/a7me0f/alma9921650780605762. 2024.

[29] Jeff Parsons. *Crate rangemap*. https://docs.rs/rangemap/latest/rangemap/. 2024. (Visited on 03/21/2024).

[30] Rust. *BTreeMap*. https://doc.rust-lang.org/std/collections/struct.BTreeMap.html. 2024. (Visited on 04/05/2024).

[31] Acorn. *Acorn community*. https://github.com/acornjs/acorn. 2024. (Visited on 04/04/2024).

[32] Titus Wormer. *estree-util-to-js*. https://github.com/syntax-tree/estree-util-to-js?tab=readme-ov-file. 2024. (Visited on 04/04/2024).

[33] Postsharp. *Metalama*. https://www.postsharp.net/metalama. 2024. (Visited on 03/07/2024).

[34] Microsoft. *Visual Studio Code*. https://code.visualstudio.com/. 2024. (Visited on 03/20/2024).

[35] Microsoft. *Using Git source control in VS Code*. https://code.visualstudio.com/docs/sourcecontrol/overview. 2024. (Visited on 03/20/2024).

[36] Microsoft. *Webview API*. https://code.visualstudio.com/api/extension-guides/webview. 2024. (Visited on 04/01/2024).

[37] *Rosetta Code*. https://rosettacode.org/. (Visited on 06/07/2024).

[38] *The Computer Language Benchmarks Game*. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html. (Visited on 06/07/2024).

[39] RealWorld. *Introduction*. https://main--realworld-docs.netlify.app/docs/intro. 2024. (Visited on 04/18/2024).

[40] Postman. *Run and test collections from the command line using Newman CLI*. https://learning.postman.com/docs/collections/using-newman-cli/command-line-integration-with-newman/. 2023. (Visited on 04/18/2024).

[41] Postman. *What is Postman? Postman API Platform*. https://www.postman.com/product/what-is-postman/. 2024. (Visited on 04/18/2024).

[42] Moritz Lipp et al. "PLATYPUS: Software-based Power Side-Channel Attacks on x86". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 355–371. DOI: 10.1109/SP40001.2021.00063.

[43] Jordan Shamir. *5 benefits of virtualization*. URL: https://www.ibm.com/think/insights/virtualization-benefits#:~:text=The%20consolidation%20of%20the%20applications,cost%20savings%20to%20your%20organization..

[44] *Container benefits*. URL: https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/container-benefits.html.

[45] AWS. *Recent Software-based Power Side-Channel Security Research*. https://aws.amazon.com/security/security-bulletins/AWS-2023-005/. (Visited on 05/07/2024).

[46] Eddie Antonio Santos et al. "How does docker affect energy consumption? Evaluating workloads in and out of Docker containers". In: *Journal of Systems and Software* 146 (2018), pp. 14–25. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.07.077. URL: https://www.sciencedirect.com/science/article/pii/S0164121218301456.

[47] Jóakim v. Kistowski et al. "Analysis of the Influences on Server Power Consumption and Energy Efficiency for CPU-Intensive Workloads". In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: Association for Computing Machinery, 2015, 223–234. ISBN: 9781450332484. DOI: 10.1145/2668930.2688057. URL: https://doi.org/10.1145/2668930.2688057.

[48] *The top programming languages*. URL: https://octoverse.github.com/2022/top-programming-languages.

[49] *Scaphandre*. https://github.com/hubblo-org/scaphandre. (Visited on 05/30/2024).

[50] *NDepend*. URL: https://marketplace.visualstudio.com/items?itemName=PatrickSmacchia.NDepend.

[51] *About self-hosted runners*. URL: https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/about-self-hosted-runners.

[52] *OAuth 2.0*. https://oauth.net/2/. (Visited on 06/04/2024).

[53] *What is SSH Public Key Authentication?* https://www.ssh.com/academy/ssh/public-key-authentication. (Visited on 06/04/2024).

[54] *What is MITM (Man in the Middle Middle) Attack*. https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/. (Visited on 06/04/2024).

[55] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[56] dnSpyEx. *dnSpyEx*. https://github.com/dnSpyEx/dnSpy. (Visited on 04/22/2024).

Appendix A

Stubbing of micro-benchmarks

As mentioned in Section 5.2.1, we used the method introduced by Jagroep et al.[4] in comparison with Thor. This section will explain how the benchmarks were stubbed. The code can be found on our repository [1].

The benchmarks are in the form of functions, these are replaced by stubbed versions. As N-body and Fibonacci return numbers they use the same stubbed function that returns 0. The sorting benchmarks MergeSort and QuickSort are in-place, which means that the list given as an argument is sorted and not copied into a new sorted list. The functions used for stubbing can be seen in listing A.1 and A.2.

```
function stubbed(input) {
    return 0;
}
```

Listing A.1: Stubbed function used for N-body and Fibonacci.

```
function stubbed(input) {
    return input;
}
```

Listing A.2: Stubbed function used for MergeSort and QuickSort.

[1] https://github.com/cs-24-pt-10-01/HotspotBenchmarkJS/tree/Guzzler_method

Appendix B

Excessive recursion

This appendix explains how the amount of recursive calls from Fibonacci is a problem for Thor. While the focus is on Fibonacci, other programs can also have this problem.

The implementation of Fibonacci used in the test, described in Section 5.2.1, uses recursive calls to calculate the desired number of the Fibonacci sequence, which, in this case, is 47. The Fibonacci function can be seen in Listing B.1.

```
function fib(n) {
  return n < 2 ? n : fib(n - 1) + fib(n - 2);
}
```

Listing B.1: Fibonacci function from Rosetta code

Using 47 as the input results in 9,615,053,951 calls of the `fib` function. The example packet from a start call, seen in Appendix E, is 290 bytes. Assuming each start and stop call results in a packet with a similar size, it would result in 5.576 terabytes. This amount of data is more than what we consider reasonable for a system such as Thor to produce.

Appendix C

Reverse Engineering Metalama code

To give an insight into what C# code Metalama generates at compile-time, the resulting compiled binary is analyzed with a reverse engineering tool and compared. The chosen tool for analyzing the code is dnSpyEx[56].

An example of a C# function from the RealWorld C# implementation is shown in Listing C.1. It is the CreateAsync function in the UserHandler class, used for creating a user.

```
1 public async Task<UserDto> CreateAsync(NewUserDto newUser,
      CancellationToken cancellationToken)
2 {
3     var user = new User(newUser);
4     await _repository.AddUserAsync(user);
5     await _repository.SaveChangesAsync(cancellationToken);
6     var token = _tokenGenerator.CreateToken(user.Username);
7     return new UserDto(user.Username, user.Email, token, user.Bio,
      user.Image);
8 }
```

Listing C.1: C# RealWorld CreateAsync implementation

The generated code at compile-time is different from the default implementation, seen in Listing C.1. The decompiled code can be seen in Listing C.2. The call to the `CreateAsync_Source` function, seen on line 4 in Listing C.2, has been added. The `CreateAsync_Source` function can be seen as the original function with the `CreateAsync` function now being the function containing the measurement code. The code generated by Metalama is able to measure everything between the beginning and the end of the function. However, it does not measure inside the added `CreateAsync_Source` function, meaning that a deeper perspective of all subfunction calls are not gathered. Therefore, only a broad overview of the whole function's energy usage is gathered.

It was found that the code generated by Metalama is equivalent to what is expected.

```
public async Task<UserDto> CreateAsync(NewUserDto newUser,
    CancellationToken cancellationToken)
{
    global::Thor.Thor.start_rapl("CreateAsync");
    UserDto userDto = await this.CreateAsync_Source(newUser,
    cancellationToken);
    UserDto result = userDto;
    global::Thor.Thor.stop_rapl("CreateAsync");
    return result;
}

private async Task<UserDto> CreateAsync_Source(NewUserDto newUser,
    CancellationToken cancellationToken)
{
    User user = new User(newUser);
    await this._repository.AddUserAsync(user);
    await this._repository.SaveChangesAsync(cancellationToken);
    string token = this._tokenGenerator.CreateToken(user.Username);
    return new UserDto(user.Username, user.Email, token, user.Bio,
    user.Image);
}
```

Listing C.2: Decompiled C# RealWorld CreateAsync Metalama generated code

Appendix D

Additional Workload

As explained in Section 5.2.1, a test is conducted where an additional workload is added to change the primary hotspot of the micro-benchmark mix test. This section explains how the amount of additional workload was chosen.

Deep cloning is used to add the additional workload, as it does not change the input. As a single deep clone does not add the necessary workload, the input of MergeSort is cloned multiple times. To find the amount needed for the desired workload, we executed MergeSort with a varying amount of deep clones. The executions can be seen in Table D.1. From the results presented in Section 6.1, it can be seen that the primary hotspot Fibonacci used 1054.76 joules on average. When the input of MergeSort is cloned 30,000 times, MergeSort yields a higher energy consumption of 1131.98 joules on average.

Deep clone count	Accumulated	Average
10000	4230.09	423.01
15000	6213.91	621.39
20000	7898.93	789.89
25000	9521.32	952.13
30000	11319.81	1131.98

Table D.1: Energy consumption of 10 executions of MergeSort with Additional deep clone with a varying count.

Appendix E

Packet representations

This section shows the format of the data sent from the process-under-test to Thor and from Thor to the client.

E.1 Process-under-test packet

An example of data, sent from the process-under-test, can be seen in Listing E.1. A breakdown of this data can be seen in Table E.1.

```
26, 00, 00, 00, 00, 00, 00, 00, 31, 3A, 2E, 2F, 48, 6F, 74, 73,
70, 6F, 74, 42, 65, 6E, 63, 68, 6D, 61, 72, 6B, 4A, 53, 2F, 6D,
61, 69, 6E, 2E, 6A, 73, 3A, 72, 65, 71, 75, 69, 72, 65, DF, 06,
00, 00, 40, E8, C5, C3, 13, 7F, 00, 00, 00, 00, 00, 00, 64, E3,
CB, 78, 98, EB, D3, 17, 00, 00, 00, 00, 00, 00, 00, 00
```

Listing E.1: A packet sent from the process-under-test represented as hexadecimal values

Field name	Value	Bytes
Length of Id	38	26, 00, 00, 00, 00, 00, 00, 00
Id	1:./HotspotBenchmarkJS/main.js:require	31, 3A, 2E, 2F, 48, 6F, 74, 73, 70, 6F, 74, 42, 65, 6E, 63, 68, 6D, 61, 72, 6B, 4A, 53, 2F, 6D, 61, 69, 6E, 2E, 6A, 73, 3A, 72, 65, 71, 75, 69, 72, 65
Process id (32 bit unsigned)	1759	DF, 06, 00, 00
Thread id (64 bit unsigned)	139722865633344	40, E8, C5, C3, 13, 7F, 00, 00
Operation (enum)	Start	00, 00, 00, 00
Timestamp (128 bit unsigned)	1716974923052475236	64, E3, CB, 78, 98, EB, D3, 17, 00, 00, 00, 00, 00, 00, 00, 00

Table E.1: Breakdown of the content within the packet sent from the process-under-test.

E.2 Client packet

After receiving a packet from the process-under-test, the server then matches the packet's timestamp with a RAPL measurement. This results in a packet for the client which is serialized to JSON. A serialized example can be seen in Listing E.2. The size of the serialized packet is 290 bytes. Packets such as this will be sent in batches in the form of lists to the client.

```
{
    "process_under_test_packet": {
        "id": "1:./HotspotBenchmarkJS/main.js:require",
        "process_id": 15647,
        "thread_id": 139632949852224,
        "operation": "Start",
        "timestamp": 1716202972939155200
    },
    "rapl_measurement": {
```

```
        "Intel": {
            "pp0": 149062.57501220703,
            "pp1": 0,
            "pkg": 35349.74530029297,
            "dram": 0
        }
    },
    "pkg_overflow": 0
}
```

Listing E.2: A single packet to the client serialized to a JSON string.

Appendix F

Stubbed RealWorld implementations

The RealWorld web service, explained in Section 5.2.2, is a more realistic program than the micro-benchmark mix, which is explained in Section 5.2.1. Using Jagroep et al.'s method[4] on the RealWorld implementation and comparing the results to those produced by Thor would further the confidence in Thor's ability to detect hotspots.

Since stubbing requires time, and the RealWorld implementations contains many functions, it was decided to only stub the functions which Thor reported as the ten most energy consuming functions. These can be seen in Table 6.6, and 6.7, for JavaScript and C# respectively.

Stubbed JavaScript implementation

An issue which was encountered when trying to stub the 10 most energy consuming functions for JavaScript is that not all of the identified hotspots can easily be stubbed. The JavaScript implementation of the RealWorld web service contains multiple `require` functions within the functions with the highest energy consumption. The `require` function is used to import code from other sources. Since the imported functionality is used by other parts of the implementation, removing the `require` is not possible without breaking the program.

Because of these issues, only three functions of the ten most expensive were stubbed in the JavaScript implementation. Only a single hash function was stubbed, as both hash functions seen in Table 6.6 result in the execution of the same code. The results from 10 executions with stubbing can be seen in Table F.1. Compared to the results of Thor, the placement of the `hash` and `compare` functions have switched places, as `compare` is measured to have a bigger impact on energy consumption.

An issue encountered when trying to interpret the results is high variance of the energy consumption for the whole program. This variance can be seen in Figure F.1, where the difference between the maximum and minimum energy consumption is around 60 joules. This difference is larger than the consumption of the stubbed functions. This variance can have a larger effect than the stubbed functions and thereby change which function is identified as the hotspot.

Function stubbed	**Accumulated [Joules]**	**Average [Joules]**
No function stubbed	1718.20	171.82
getArticles	1717.60	171.76
hash	1603.11	160.31
compare	1489.91	148.99

Table F.1: The energy consumption from 10 executions of the JavaScript implementation with and without stubbing.

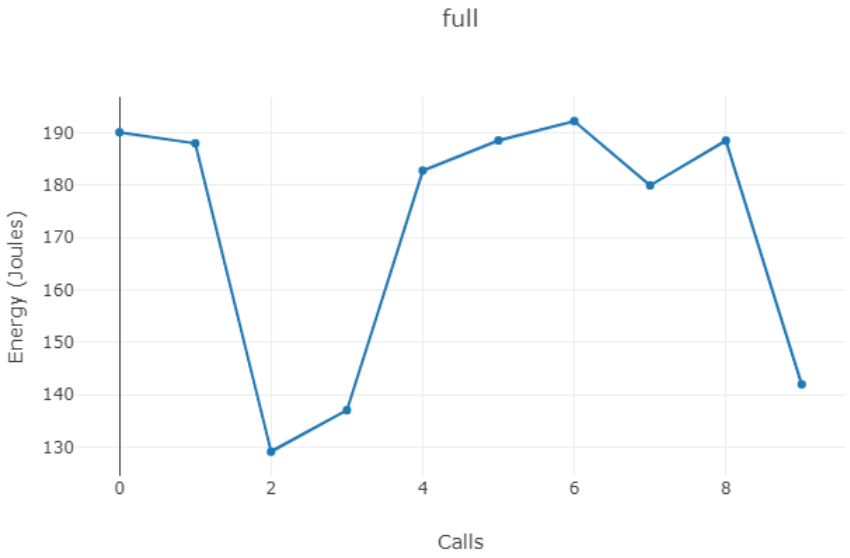


Figure F.1: The energy consumption of 10 executions of the JavaScript implementation.

The issue with the inter-dependencies could be avoided, if the entire component is stubbed. As an example, the `run` function seen in Table 6.6, is a function which starts the database. Stubbing just the `run` function would stop code dependent on the database from working. If the entire database was stubbed, including each database function, then any part of the program which interacts with the database could receive a mocked response. By stubbing entire components, the granularity

that hotspots can be detected at is reduced. Jagroep et al.'s method were also tested on components and not individual functions in their own work[4].

By testing on whole components, the energy consumption for the component will be higher, and the variance will thereby have a lesser impact.

Stubbed C# implementation

We also wanted to use Jagroep et al.'s method on the C# implementation. The functions identified by Thor as the 10 most energy consuming functions are all able to be stubbed, with varying degrees of work needed. Before we began testing with stubbed versions, we ran the non-stubbed version 10 times to get the variance between runs.

The variance can be seen in Figure F.2. It can be seen that the difference between the maximum and minimum energy consumption is around 44 joules. The same difference is expected to be present in the stubbed versions. Because of the difference of around 44 joules, the results from testing the stubbed versions are likely going to vary significantly.

If we look at the energy consumption of the hotspots provided by Thor, which can be seen in Table 6.7, it can be seen that the most energy consuming hotspot, when not counting `main`, is the `CreateAsync` method of the `UserHandler` class with an energy consumption of 13.35 joules.

Because the results from using Jagroep et al.'s method can have a difference higher than the reported energy consumption by Thor, comparing the results will likely be inconclusive. Therefore, we decided not to test the stubbed variants and compare them with Thor.

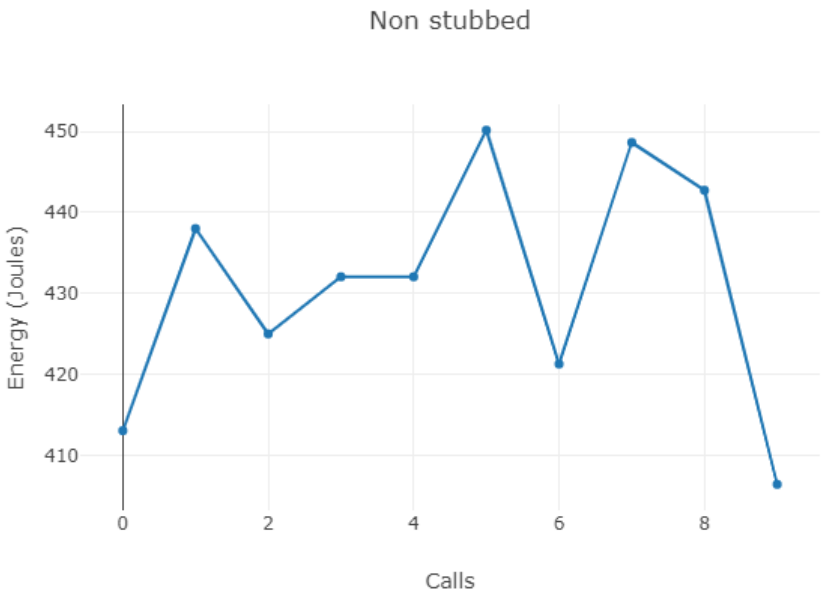


Figure F.2: Variance of the energy consumption of C# Realworld implementation with no stubbing