



Universität Stuttgart

Institut für Architektur von Anwendungssystemen

Universitätsstraße 38  
70569 Stuttgart

**Masterarbeit**  
**Continuous Estimation of Energy**  
**Efficiency**  
**for Source Code in Virtual**  
**Environments**

Maximilian Niklas Schulth

**Studiengang:** Master of Science Softwaretechnik

**1. Prüfer:** Prof. Dr. Marco Aiello

**2. Prüfer:** Prof. Dr. Dr. h. c. Frank Leymann

**Betreuer:** Dinesh Vermula

**begonnen am:** 01.05.2024

**beendet am:** 01.11.2024

# Acknowledgements

First of all I would like to thank Prof. Dr. Marco Aiello and TeamViewer for making this thesis possible in the first place and for the financial support TeamViewer provided to me.

Also, I would like to thank my supervisor at TeamViewer, Fabian Pachner, for his continued support and his time to give me valuable advice on my ideas.

## Abstract

The increasing energy consumption of servers, high-performance computing clusters, and data centers necessitates measures to reduce energy consumption. However, many companies, like TeamViewer, rely on rented infrastructure where direct hardware-level energy management is unavailable. This thesis presents a method for estimating software energy efficiency on virtualized server environments, focusing on optimizing code execution without access to physical hardware metrics.

The research addresses several key challenges, including how to measure energy consumption at the function level of a software, simulate realistic and reproducible user loads, and considerations for isolating performance measurements from external influences. Profiling tools were evaluated to measure CPU time, a metric which is correlated with energy consumption. The method was tested in a virtual environment by simulating user loads and measuring the impact of software changes on performance.

The results demonstrate that CPU time can provide insights into the performance of a software which correlates with its energy consumption. This work contributes to the field by providing a lightweight method for continuously estimating the energy efficiency during software development and maintenance.

## Kurzfassung

Der steigende Energieverbrauch von Servern, Hochleistungsrechnerclustern und Rechenzentren erfordert Maßnahmen zur Reduzierung des Energieverbrauchs. Viele Unternehmen, wie TeamViewer, sind jedoch auf gemietete Infrastrukturen angewiesen, bei denen ein direktes Energiemanagement auf Hardware-Ebene nicht möglich ist. Diese Arbeit stellt eine Methode zur Abschätzung der Energieeffizienz von Software in virtualisierten Serverumgebungen vor, mit dem Fokus auf die Optimierung der Codeausführung ohne Zugriff auf physische Hardware-Metriken.

Die Forschung befasst sich mit mehreren zentralen Herausforderungen, darunter die Frage, wie der Energieverbrauch auf Funktionsebene einer Software gemessen, realistische und reproduzierbare Benutzerlasten simuliert und Überlegungen zur Isolierung von Leistungsdaten vor äußeren Einflüssen angestellt werden können. Es wurden Profiling-Tools evaluiert, um die CPU-Zeit zu messen, eine Metrik, die mit dem Energieverbrauch korreliert. Die Methode wurde in einer virtuellen Umgebung getestet, indem Benutzerlasten simuliert und die Auswirkungen von Softwareänderungen auf die Leistung gemessen wurden.

Die Ergebnisse zeigen, dass die CPU-Zeit Einblicke in die Leistung einer Software geben kann, die mit ihrem Energieverbrauch korreliert. Diese Arbeit leistet einen Beitrag zum Fachgebiet, indem sie eine einfache Methode zur kontinuierlichen Abschätzung der Energieeffizienz während der Softwareentwicklung und -wartung bereitstellt.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Goals and Research Questions . . . . .	1
1.2	Results of this Thesis . . . . .	2
1.3	Thesis Outline . . . . .	2
1.4	TeamViewer . . . . .	3
<b>2</b>	<b>Fundamentals</b>	<b>4</b>
2.1	Static and Dynamic Program Analysis . . . . .	4
2.2	Profilers . . . . .	4
2.2.1	Sampling Profiler . . . . .	4
2.2.2	Instrumentation Profiler . . . . .	4
2.3	Energy Consumption or Power Consumption . . . . .	5
2.4	Analyzing at the Function Level . . . . .	5
2.5	External Noise during the Measurement . . . . .	5
<b>3</b>	<b>Related Work</b>	<b>6</b>
<b>4</b>	<b>Design of Solution</b>	<b>8</b>
4.1	Metric Selection . . . . .	8
4.2	Measurement Method . . . . .	9
4.2.1	Codetrack . . . . .	9
4.2.2	Very Sleepy . . . . .	10
4.2.3	Perfview . . . . .	10
4.2.4	Jetbrains dotTrace . . . . .	10
4.2.5	Building a Customized Profiler . . . . .	11
4.2.6	Comparison Summary . . . . .	11
4.3	Isolation of Measurement . . . . .	12
<b>5</b>	<b>Implementation</b>	<b>13</b>
5.1	Simulating User Load Through Load Testing . . . . .	13
5.2	First Measurement and Verifying the Measurement Setup . . . . .	16
5.3	Measurement of degrading performance . . . . .	21
<b>6</b>	<b>Discussion</b>	<b>23</b>
<b>7</b>	<b>Conclusion and Future Work</b>	<b>24</b>

# 1 Introduction

Energy consumption of servers, high-performance computing clusters, and data centers is continuously increasing, due to growing computational demands. Because of global warming, there is also a growing demand for saving energy by reducing energy consumption and minimizing energy waste as much as possible.[1]

To effectively reduce energy consumption, it is essential to identify areas where optimizations can be made. While measuring energy consumption on a bare metal server with access to the operating system has been researched to a great extent, as can be seen in the survey by Dayarathna, Wen, and Fan [2], not every software company possesses its own IT infrastructure to host distributed software products. This is due to the significant additional effort and resources required to set up and maintain the infrastructure, diverting focus from core business activities. As a result, many companies decide to rent infrastructure from providers. However, this limits their direct ability to influence saving energy on hardware, leaving downscaling or switching providers as two of the last few options.

An alternative approach to minimize energy consumption is to optimize the software running on the infrastructure itself, once it is in an executable state. Unfortunately, directly measuring the energy consumption of individual software components within a system presents challenges. Software energy consumption is often intertwined with that of other applications, making precise measurement difficult. Furthermore, the energy consumption of software is influenced by the hardware it operates on, meaning the same software may exhibit varying energy consumption levels on different hardware configurations.[3]

## 1.1 Goals and Research Questions

The main focus of this thesis is to find a suitable method for estimating software energy efficiency in virtual machines (VM), without incorporating any physical hardware metrics. Consequently, it is important to investigate other possibilities of how the energy consumption of a software product can be measured. This requires looking at different aspects systematically by asking the following questions.

**RQ1.1: Which metric correlates with the energy consumption of the software and is measurable in a VM?**

Due to the exclusion of access to the hardware of the server, it is important that the measurement results can be obtained in the first place. The obtained data should closely match the data of the actual energy consumption. This requires the analysis of suitable software metrics, taking into consideration the limitations inside of a VM to find the optimal one.

**RQ1.2: How can load be induced on the system, which is reproducible and at the same time represents a real world usage scenario?**

In order to compare measurements with each other, they need to be created as similar as possible. In the best case they are identical. To achieve this there needs to be a predefined set of operations that create a load on the system which is deterministic and reproducible. Furthermore, it is important to figure out how to create a realistic load scenario.

**RQ1.3: How can the measurement be isolated from other influencing factors?**

Due to the fact that during daily use, the server is occupied by other users. This external load could influence the measurement data. Additionally, maintenance and other scheduled operations, such as updates or deployments may also affect the data.

**RQ1.4: How can the energy consumption of the software be analyzed on a function-level to find the code that can be optimized?**

We want to be able to optimize code. If we only measure the whole application it is hard to guess where to improve. This is why we need to focus on measuring at the function level.

**RQ1.5: How to make the measurement comparable with measurements at different times?**

To make the measurements comparable there needs to be some score that also can be used even when new features get added or old features get removed. New features with an already existing and measured feature may skew the measurement score.

**RQ1.6: How can the results be evaluated to make sure they correlate with the real world energy consumption?**

Estimating the energy consumption instead of measuring means that the results need to be evaluated of how accurate they actually are and if they correlate with the actual energy consumption of the software.

**RQ1.7: How can the tests be integrated so that they can be executed automatically in a recurring manner?**

The measurements can be used to compare the change in energy consumption of certain software components over time. For this to be effective, the tests should be easy to start or ideally run automatically.

## 1.2 Results of this Thesis

This thesis contributes a practical approach to estimating software energy efficiency in VM's. By focusing on CPU time as a metric, which we assume to be closely correlating with energy consumption, the work enables a more precise understanding of software performance in the absence of direct hardware measurements. While the method presented is not perfect, it offers a lightweight and continuous monitoring solution that can be integrated into a continuous integration and continuous deployment (CICD) pipeline.

Another contribution is an approach to create realistic load test scenarios, which are created from the recorded network data of the browser. This allows to create scenarios by recording network activity with just the browser and the load test script. The load tests can even be used if an authentication measure is in place.

The results highlight a clear correlation between the total runtime the CPU spends inside of a function and software performance, allowing targeted optimization of software code. However, the experiments were conducted under controlled conditions on specific hardware setups, which may limit their applicability to other environments. Future work should explore further validation across different configurations and workloads to refine the approach.

Overall, the thesis contributes to the field of improving software energy efficiency, while acknowledging the limitations of the method and areas for potential improvement.

## 1.3 Thesis Outline

This paper is structured as follows: Section 2 provides a theoretical background of key techniques used for conducting the data collection. In Section 3, an overview is provided of related research on this topic showing various different approaches and how they differ from this work. This then leads to Section 4 in which the procedure used to extract the data is explained. Based on the criteria mentioned in Section 1.1, different profilers are highlighted and compared. Section 5 presents the steps taken to conduct the experiment as well as the results. In Section 6 the potential biases and measurement errors, or in other words, the limitations of the experiment are discussed. Lastly, Section 7 concludes the findings

and gives an outlook on how further research could be conducted to gain more insight into the energy consumption estimations.

## **1.4 TeamViewer**

TeamViewer is a company which has its headquarters located in Göppingen, Germany. They developed and maintain the remote control software "TeamViewer" and have also expanded by developing other remote software. This thesis was written in the business unit Remote Management of the Research and Development department. This thesis was created during a paid master student contract with TeamViewer for 6 months. The topic for this thesis was created by the author and approved by TeamViewer and the University of Stuttgart.

## 2 Fundamentals

This section provides an explanation of the key concepts and tools used throughout this thesis. It covers the basics of static and dynamic program analysis, which are essential for identifying performance inefficiencies in software. Additionally, the section clarifies why energy consumption instead of power consumption is used, helping to prevent misunderstandings.

### 2.1 Static and Dynamic Program Analysis

Static program analysis refers to the process of examining source code without executing it, typically using automated tools to detect potential issues such as coding errors, security vulnerabilities, and adherence to coding standards. It involves analyzing the code's structure, syntax, and semantics to identify bugs, inefficiencies, and maintainability concerns before the program is run.[4]

Dynamic program analysis, on the other hand, involves analyzing the program while it is executing. This method examines the program's behavior during runtime, monitoring variables, memory usage, and control flow to detect errors, memory leaks, and performance bottlenecks. Dynamic analysis provides insights into how the program operates in real-world scenarios, helping developers identify and address issues that may not be apparent during static program analysis.[5]

### 2.2 Profilers

One way of performing dynamic program analysis is by using software profilers. Developers often utilize software profilers, which can be attached to their software to find inefficient code during development or even maintenance once the software is in an executable state. These profilers gather metrics that help identify performance bottlenecks and areas for improvement within the software. There are different ways to profile software, some of the more well known types will be explained in the following subsections.

#### 2.2.1 Sampling Profiler

The following information is based on an article of the website pixie blog.<sup>1</sup> A sampling profiler is a tool used in software development to analyze the performance of a program by periodically sampling its execution state. Rather than tracking every single function call or instruction, a sampling profiler periodically interrupts the program's execution at predefined intervals and records the current stack trace. By sampling the program's state at regular intervals, the profiler builds a statistical representation of where the program spends its time, identifying hot spots or frequently executed code paths.

Sampling profilers are lightweight and have minimal impact on the program's performance, making them suitable for use in production systems. They provide insights into the runtime behavior of the program, highlighting areas of inefficiency or performance bottlenecks that may require optimization.

A limitation of sampling is that its statistical nature always includes a measurement error. Smaller functions might not even show up in the report because the sample did not hit it directly.

#### 2.2.2 Instrumentation Profiler

An instrumentation profiler inserts code into the translated application code (Assembly) of a program to gather information every time a function is entered or exited. Yet, this can add a lot of overhead, because

---

<sup>1</sup><https://blog.px.dev/cpu-profiling/> last accessed: 14th of October 2024

the instrumentation code is executed for even the smallest function call, thus it is not suitable to profile software running in production systems as it would slow them down and sacrifice valuable resources.<sup>2</sup> This type of profiler has its strength in its simple setup and thorough measurement. As it measures every singly line of executed code, there are no parameters needed to fine tune the measurement. The thorough measurement can be used to tell exactly how many times any function was called, which is not possible with the sampling profiler.

## 2.3 Energy Consumption or Power Consumption

The term "power consumption" is frequently coined by the academic community in the papers referenced in this thesis.[6, 7, 8, 9, 10, 11] Technically, the term power refers to the amount of work done at any moment, not counting in the time. Energy measures the total quantity of work over time. Therefore, "energy consumption" is the more appropriate term to use here.<sup>3</sup>

## 2.4 Analyzing at the Function Level

The level of detail is an important aspect during code or software analysis. A software can be analyzed as a whole or in various levels of detail. We could for example go all the way down to analyzing the assembler instructions but this would only make sense for an appropriate use case. For this thesis it is most useful to look at a level of detail in which we can differentiate between the execution times of different functions. Therefore, whenever we want to talk about this level of detail during analysis we will reference it as "at the function level".

## 2.5 External Noise during the Measurement

Whenever we mention external noise, we are referencing to the external influences on our measurement. The term "external noise" refers to anything that is outside of our direct influence and affects our measurement results and are hard to predict. External noise can be other users on the system who create load on the system which we then measure. Other forms of external noise could be updates on the system which are triggered externally or even a crash of one or more parts of the system which would result in very obscure or no data at all.

---

<sup>2</sup><https://learn.microsoft.com/en-us/visualstudio/profiling/understanding-performance-collection-methods-perf-profiler?view=vs-2022#instrumentation> last accessed: 14th of October 2024

<sup>3</sup><https://www.blackhillsolar.com/post/energy-and-power> last accessed: 14th of October 2024

### 3 Related Work

In order to gain an overview of the current state of research on the topic of this thesis, related work was reviewed. The process of finding related work consisted of two main steps. First, the most renowned libraries for computer science were searched with the same or very similar keywords from the title of this paper, e.g. "consumption", "server", "efficiency", and "measur\*(e|ing|ement)" (Some search engines use different syntax or need more filtering for more relevant results). This resulted in 326 results for the IEEE library, 13 results for Sciencedirect, 53 results for the Information Systems Applications library, and 100 results for the ACM library. Then, non related papers were filtered out by reading the implementation and conclusion parts to see what the paper was about. Most of the papers focused on bare-metal systems and metrics that require direct access to the hardware for measuring the energy consumption, from which we only present a small selection.[6, 7, 8] Therefore, of the 492 papers, only 34 papers remained, from which 8 were selected as relevant to this thesis and their significance is explained in the following paragraphs

Wu et al. [6] created an energy consumption model with an Elman Neural Network (PCM-ENN) to estimate real-time energy consumption without the need for any extra metering devices. They trained the model on datasets of mixed workloads (CPU-intensive, Memory intensive and I/O intensive). The input to the model is gathered through performance counters of CPU utilization, memory usage, disk throughput and disk IO request rate. The results from their experiments show that PCM-ENN estimated results with mean relative errors (MRE) of less than 4% and 2% on two different server datasets. The model outperformed other methods like linear regression and NARX neural networks in terms of accuracy. However, due to the use of performance counters this model cannot be used in VM's in its current form as they are not available in the virtualized abstraction layer. While Wu et al. propose PCM-ENN to accurately estimate energy consumption based on time-series data, our approach employs CPU-time as an indirect metric for energy consumption in VM's, and assuming no access to detailed hardware metrics. Also, the PCM-ENN model is an end-to-end black box energy consumption model, which does not measure at the function level but at the system level.

In their paper, Li et al. [12] present a method to estimate energy consumption at an architecture level by modeling the software as a network. The interactions are modeled as edges and software entities are modeled as nodes. They then train a machine learning model using benchmarks on a test system. The model is claimed to achieve an error rate of 7.9%. In comparison to our measurement their estimation seems to focus less on a real usage scenario for the creation of load on the system. Thus, since the model is trained on artificial benchmarks and the current software state it could lead to a more significant margin of error in real scenarios or if the software changes. In comparison to the work of Li et al., which estimates the energy consumption based on static code analysis of characteristics at an architecture level, our work tries to measure each function dynamically during runtime.

Zhu et al. [7] created a Gaussian Mixture Model to estimate energy consumption at different usage levels. While their model achieves lower root mean squared error (RMSE) and higher success ratios than linear and other neural network-based models, it requires significantly more training time and a large dataset to ensure accuracy. Their model is made to be used in cluster environments like hadoop. Unfortunately, in the paper it is not explained if the model is limited only to servers and clusters with direct access to the operating system metrics, which is why we assume it cannot be applied to VM's. In contrast to the long training time and large dataset, our approach is more light-weight by not needing to train a model. However, it could lack in accuracy compared to the model of Zhu et al. as we rely only on one usage levels for our load tests.

Wedyan, Morrison, and Abuomar [13] proposed an approach to estimate energy consumption of

software using unit testing by measuring the method's size, complexity and dependencies. They use the cyclomatic complexity which is calculated based on the decision points in the code, call graphs for dependencies and lines of code for the method's size for their static code analysis. However, they have not yet implemented and evaluated their approach. In comparison to our work they are not limited by the boundaries of a VM.

The work of Li et al. [9] proposes a model to meter the energy consumption of an entire VM based on the resource values of the CPU, memory, and disk usage. They apply a linear regression model to these values to get a base model and then improve it by training a sub-classified piecewise model which considers the impact of multiple VM's running on the same system at the same time. The training data for the improved model include the actual energy value which is collected by an energy meter. They claim that this allows them to reach a high average estimation accuracy of more than 96%. Despite that, due to the dependency on the energy meter this approach is limited to environments with a direct access to the hardware. Additionally, it only measures at a VM level but not at the function level to find areas for optimization. Yet, their work is very similar to ours and we could improve our approach by verifying our results through the use of an energy meter and could also look at other energy consumption proxies like memory and disk usage.

Schmitt et al. [8] present an approach to determine the energy consumption at the function level of an application, which is also our goal. Their approach does not require any changes to the source code because they utilize global performance counters and concurrently track the stack trace through the use of an already existing software called Kieker. Their results show a strong correlation between the measured CPU-time and the actual energy consumption measured through an energy meter. Our work builds greatly on top of their findings as we assume the CPU-time to be a reasonable proxy for the energy consumption of the software.

Dhiman, Mihic, and Rosing [10] present a system for online energy consumption prediction in virtualized environments using a Gaussian Mixture Model. Their approach dynamically estimates the energy consumption of both physical machines and individual VM's by analyzing architectural metrics such as instruction throughput and memory access rate. This method achieves an average prediction error of less than 10%, significantly outperforming regression-based models. Their model is non-intrusive and scalable across different machine configurations. Nevertheless, their approach focuses on measuring the energy consumption of entire VM's instead of measuring at the function level.

Kansal et al. [11] trained a power model to estimate the energy consumption in VM's with the name Joulemeter. It monitors the energy consumption of the entire VM using the CPU utilization, last-level cache misses (memory), and disk input and output operations. An initial power model is trained on the measurement data when the server is not running as a VM. With Joulemeter and power capping they were able to save 13% to 27% of energy consumption. Similar to the work of Dhiman, Mihic, and Rosing, the work of Kansal et al. measures at a VM level instead of the function level.

Most of the papers have similar goals of measuring energy consumption, however, they either do not consider VM's or they do not measure at the function level, but do not consider both. The paper that is most relevant to our work is the research conducted by Schmitt et al. who provide the base for our assumption that the CPU-time is a closely correlating proxy to the energy consumption of the software. However, to the best of our knowledge, we discovered no actual approach to measuring the energy consumption of software at the function level inside of VM's in any of the research papers we examined.

## 4 Design of Solution

This section outlines the approach used to address the research questions identified earlier, focusing on the continuous estimation of software energy efficiency in VM's. Moreover, it will explain the decision making behind the selection of metrics and tools, describe the chosen profiler, and explain the necessary steps to ensure the isolation of measurements from external noise.

### 4.1 Metric Selection

Research question **RQ1.1** is about a metric that correlates with the energy consumption of the software being measured. If there was access to the hardware of the server, the energy consumption could be measured using an energy meter. With such an energy meter we would measure the system once in an idle state without the software running and once with the software running under load. The energy consumption of the software would then be the energy consumption of the system with the software under load minus the energy consumption of the system in an idle state.

$$E_{Software} = E_{SystemLoad} - E_{SystemIdle}$$

Since we do not have access to the hardware, using an energy meter is not an option. Additionally, the limited access to underlying resources within a VM makes it impossible to directly measure the absolute energy consumption of the hosting hardware without further external data or measurements. Schmitt et al. explain that in virtualized environments the performance events can be recorded on the host system and then dynamically mapped from each CPU core to the VM, to make them available inside of the VM.[8] However, this means that we need access to the underlying system yet again, which makes this not suitable for us, but could be useful to know for other use cases.

The primary goal of this thesis is to identify a method for measuring differences in energy consumption between different software states, with the aim of improving energy efficiency. As a result, the absolute energy consumption value is not as important as the relative difference in energy consumption between two software states. Given this, we can reasonably assume that measuring a metric that closely correlates with the system's actual energy efficiency will be sufficient to meet our goal.

In Table 1 we compare some of the metrics that are mentioned in related works to give an overview of which would be a viable choice for our work. A metric that closely correlates with the energy consumption of the system is the time that the CPU spends in a non-idle state. For a software this would be the time the CPU spends processing its instructions. Using it as a metric for energy consumption is controversial as it does not always correlate directly with the actual energy consumption. As identified by Kansel et al. the CPU-time correlates with the energy consumption but with a non-trivial margin of error.[11] Similarly, Beyer and Wendler conclude in their article from 2020 that CPU-time alone is not sufficient to estimate energy consumption and propose using Intel's running average power limit (RAPL).[14]

Also, it can definitely be measured inside of a VM. Therefore, it receives an x-mark for accuracy and a checkmark for being able to run in a VM in Table 1.

With the built-in performance counters, Schmitt et al. got measurements with an accuracy of 7.5% for light load and 2.6% for heavy load.[8] Nevertheless, RAPL and Performance Counters depend on direct access to the CPU cores. Direct access to the hardware is not possible in VM's by default. There are ways to enable performance counters in some VM's, however, it is not possible in all of them<sup>4</sup> and

---

<sup>4</sup>[https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.vm\\_admin.doc/GUID-F920A3C7-3B42-4E78-8EA7-961E49AF479D.html](https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.vm_admin.doc/GUID-F920A3C7-3B42-4E78-8EA7-961E49AF479D.html)

Criterion	CPU-Time	RAPL	Performance Counters
Accuracy	✗	✓	✓
Runs in VM	✓	✗	✗

Table 1: Comparison of metrics for measuring the energy consumption.

requires direct administrative access to the host or configuration panel, which is why we conclude that performance counters and RAPL do not fulfill the requirement to run in VM's.

Despite the distrust in the correlation of CPU time and energy consumption, we will be measuring the CPU time as it is still the closest correlating metric for energy consumption which can be measured in VM's.

Despite the added accuracy of also measuring the memory and disk usage, we do not consider it in this thesis for simplicity.

## 4.2 Measurement Method

Research question **RQ1.4** deals with the question of how to analyze the energy consumption of the code at the function level. Measuring at the function level is the most important aspect of this question and acts as a criterion to filter out software tools that do not measure at the function level and would yield measurement results that lack in detail. As a result, software tools that do not fulfill this criterion are already filtered out and are not included in the comparison in Section 4.2.6.

Extending on that, research question **RQ1.5** addresses the need to compare measurements, which will be examined and presented in Table 2.

Research question **RQ1.7** focuses on automating the measurement so it can be run periodically. This requires that the tool or method must integrate with a CICD pipeline by being able to be executed from the command line interface (CLI).

Only tools that can operate within a VM will be considered, as we lack administrative access to the underlying operating system hosting the VM.

In the following sections, we assess whether existing tools meet these criteria and can be applied in our environment.

### 4.2.1 Codetrack

CodeTrack<sup>5</sup> is a performance profiling tool used for analyzing application behavior across multiple platforms, including .NET and Windows. We used version V1.0.3.3 from 10th of January 2018 to test its features and behavior. It provides metrics on CPU utilization, memory consumption, and thread performance, allowing for examination of how code runs at runtime. CodeTrack supports real-time data collection, helping to identify performance issues in both single-threaded and multithreaded applications. It also tracks memory allocations and monitors function call frequencies, giving insights into program execution. There is an option to disable processing of the measurement data on the profiled machine to make it easier performing measurements on weak systems. However, it does not feature a CLI version, which makes it unsuitable for usage in a CICD pipeline. Furthermore, Codetrack was not able to open its own non-processed output file. Due to these reasons we will not be using it for our measurements.

<sup>5</sup><https://www.getcodetrack.com/> last accessed: 14th of October 2024

### 4.2.2 Very Sleepy

Very Sleepy<sup>6</sup> is an open source software that builds on top of sleepy. We used version 0.91 from the 19th of August 2021. Very Sleepy does not need to start an application itself, in order to measure it, but can also attach to an already running application. It also measures at the function level which fits our use case and uses sampling to profile applications. Furthermore, it can be controlled from the CLI, so it can be also used in a CICD pipeline. One disadvantage is that it needs to be installed on the machine and does not offer a portable version.

Another drawback is that it has a very high impact on performance during the measurement, even when using the CLI executable. In our test it had a CPU utilization of about 90% which slowed down the system considerably and did not allow to finish the measurement in a reasonable time. Also, it did not find the program database file which means that the function names were not resolved but only their addresses shown. This would require another tool and greater effort to track down a specific function. For these reasons we will not use it for our measurement.

### 4.2.3 Perfview

Perfview<sup>7</sup> is a profiler by microsoft for windows. We used version v3.1.15 released on the 27th of August 2024. It is designed to capture and analyze event tracing data, providing detailed insights into CPU utilization, memory allocation, and thread activity. PerfView specializes in processing large sets of trace data with minimal overhead, making it suitable for analyzing complex and resource-intensive applications. The tool offers features such as flame graphs for visualizing performance bottlenecks, GC (Garbage Collection) analysis for memory management, and stack trace investigations. One disadvantage is that it captures everything on the system instead of just one process. This means it consumes a lot of storage per measurement which could be problematic for the storage capacities of the VM. Another drawback is the the lacking intuitiveness of the GUI for recording and analyzing measurements. Even though it is a powerful tool and fulfills almost all requirements, we will not choose this profiler for our measurements due to the lack of intuitiveness and storage overhead.

### 4.2.4 JetBrains dotTrace

dotTrace<sup>8</sup> is a performance profiling tool developed by JetBrains for analyzing .NET applications. We used version 777.0.0.0 which was released on the 30th of August 2024. It provides detailed insights into CPU utilization, memory consumption, and potential execution bottlenecks. dotTrace supports various profiling modes, such as sampling, tracing, and line-by-line profiling, allowing for different levels of accuracy and overhead. The tool includes visualization features such as call trees and execution paths to help identify areas of inefficient code. Additionally, dotTrace offers both real-time and post-mortem analysis, with a timeline view that tracks performance changes throughout an application's runtime. In addition to that it has an extensive documentation and a CLI version allowing it to be used in a CICD pipeline. Besides the price, dotTrace seems to be the best tools for our measurement requirements at the moment. Therefore, we will use dotTrace for our measurements.

---

<sup>6</sup><https://github.com/VerySleepy/verysleepy> last accessed: 14th of October 2024

<sup>7</sup><https://github.com/microsoft/perfview> last accessed: 14th of October 2024

<sup>8</sup><https://www.jetbrains.com/profiler/> last accessed: 14th of October 2024

#### 4.2.5 Building a Customized Profiler

For very specific use cases there is also the possibility to create a customized profiler almost from scratch. This is possible in many different ways, of which one is to use the compiler flags of the Microsoft Visual C++ (MSVC) language. The MSVC compiler allows to set compiler flags that enable injecting assembler code when the machine enters and exits functions in the profiled application. The entry and exit functions are called `_penter` and `_pexit` and the compiler options to be set are `/Gh` and `/GH`. It is up to the developer to implement `_penter` and `_pexit` as they do not yet have an implementation. This allows to essentially create a simple version of an instrumentation profiler which we can use to measure the computation time of each function by starting a timer at the entry and stopping it at the exit of a function.

Most documentation online still only consider 32-bit software with inline assembly for the assembly code. However, the MSVC x64 does not support inline assembly. So, the assembly code needs its own assembly file and made available in the C++ code as an external function call. For the assembly file to be built we need to enable MASM (Microsoft Macro Assembler) in the "Build Dependencies".<sup>9</sup> This tutorial guided us on some of the implementation difficulties in MSVC x64.<sup>10</sup>

Unfortunately, as the instrumentation code gets injected into every single function of the project, it also gets injected into the extracted assembler function call and leads to the profiler recursively injecting the instrumentation code into itself. This was not a problem with the inline assembly code in MSVC x86 code due to the fact that inline code is not considered a function call in C++. To get around this problem, the external function calls need to be extracted to another project that does not have the `/Gh` and `/GH` compiler options enabled. This can be done by creating a dynamic link library (dll) which is then utilized by the profiled software.<sup>11</sup>

Another problem we encountered is that the function `SymFromAddr` from the `dbghlp` library, which we use to resolve function pointers to human-readable function names, is not thread-safe. This means that we cannot call this function in a multi-threaded application without making it thread-safe as this would cause a segmentation fault. To make it thread-safe, we wrap it in a class which synchronises the calls to this function by allowing only a single instance of the class at a time. Also, each caller must reserve a mutex to exclusively call the function.

Unfortunately, this did not fix the problem because the TeamViewer code base seems to use some functions of the `dbghlp` library or other functions which interfere with our code and leads to an invalid read (or segmentation fault) and crashes the software. This happened even though everything seemed to work in the test environment. It would have been a great effort to search for the culprit and fix it. Therefore, in order to not lose valuable time the effort to write a customized profiler was discontinued.

#### 4.2.6 Comparison Summary

We evaluated the previously mentioned tools presented in Table 2. The criteria are the intuitiveness, price, documentation and existence of a CLI. If a criterion is satisfied, it is marked with a ✓ and if not it is marked with a ✗. The criterion "Intuitive" is evaluated by how easy it is to get some first results with the software. If there is a video or an easy to follow instruction that leads to results this is also

---

<sup>9</sup>There is a bug that if you first add the assembler files and then enable MASM, the assembler files are not included in the build. They need to be deleted and added again in this case.

<sup>10</sup><https://www.codeproject.com/Articles/800172/A-Simple-Cplusplus-Profiler-on-x> last accessed: 14th of October 2024

<sup>11</sup><https://learn.microsoft.com/en-us/cpp/build/walkthrough-creating-and-using-a-dynamic-link-library-cpp?view=msvc-170> last accessed: 14th of October 2024

considered satisfied.

Codetrack and Very Sleepy satisfy this criterion by being easy to use and providing good results. dotTrace seems to have an outdated video for the graphical user interface (GUI) but worked right away with the CLI tool. Creating a custom profiler with the MSVC compiler option is not intuitive as it requires coding first and it is not trivial to get into it.

Free of cost means that the tool can be used without paying. A free trial does not satisfy this criterion. All but dotTrace are freely available on the internet. dotTrace is also only available by buying the whole DotUltimate package which is quoted at 469€ per year in Germany as of August 2024. However, a good product that fulfills all requirements is worth a lot and a company might be willing to invest money into a product that adds a lot of value.

Only dotTrace and Very Sleepy offer a documentation that is up to date. Codetrack seems to have an outdated documentation that does not go into detail and makes it hard to get into the software. The documentation by Microsoft is only for 32 bit systems and does not show how to use it for 64 bit systems. Also, it lacks explanations and useful examples for the various options on how to use it.

Despite the cost, dotTrace was the only tool that recorded useful measurements without impacting the performance of the VM too much and was intuitive to use. As a result, we will be using dotTrace for our measurements.

Criterion	Codetrack	Very Sleepy	dotTrace	MSVC Injection	Perfvieiw
Intuitive	✓	✓	✓	✗	✗
Free of cost	✓	✓	✗	✓	✓
Up-to-date Documentation	✗	✓	✓	✗	✓
CLI	✓	✓	✓	✓	✓
Performance Impact	✗	✗	✓	✗	✓

Table 2: Comparison of different profiling options

### 4.3 Isolation of Measurement

Ensuring accurate and isolated measurements is crucial when assessing energy efficiency, particularly in virtualized server environments, where multiple users and processes can generate noise. In our context, isolation refers to minimizing interference from external factors such as other users, maintenance operations, or scheduled tasks that could affect the accuracy of our performance metrics. This need aligns with research question **RQ1.3**, which focuses on isolating the measurement environment to ensure that the software under test is not influenced.

In order to reduce external interference, one approach is to schedule measurements during low-activity periods, such as weekends or designated maintenance windows. Conducting tests during these times and comparing those measurements with measurements on a regular working day will show how much of an impact the influencing factors have or if they are negligible.

However, external noise may still be present, particularly in shared environments like VM's hosted in cloud data centers. So, to further reduce interferences, the individual interactions with the system could be monitored. This would at least make it transparent to what may have caused the noise.

Yet, this was not needed in our case as will be shown in Section 5.2 where we measured during a workday and on a Sunday and noticed only a negligible level of external noise.

## 5 Implementation

In this chapter, we describe the practical implementation of the method designed to estimate the energy efficiency of software running on virtualized server environments. The implementation focuses on addressing the research questions presented earlier, particularly how to simulate realistic user loads, measure CPU time as a proxy for energy consumption, and ensure the reproducibility and accuracy of the results.

Through this practical approach, this chapter demonstrates how energy efficiency can be estimated without direct access to physical hardware, thereby providing a solution for companies relying on third-party infrastructure.

### 5.1 Simulating User Load Through Load Testing

In research question **RQ1.2** we want to find out how to create a realistic load scenario for our measurement. A load test is needed to trigger the functions of the software on the server that is being measured. Without creating a realistic load test scenario, the measurement will not resemble how the software would function with real users in production. Furthermore, we need to be able to recreate the same load to ensure that our measurements are comparable. If we do not apply the same load for each test run, it would be impossible to determine whether a change in energy consumption for a specific function is due to a recent change or simply the result of a different load.

A straightforward approach to creating a load scenario would be to call each function n-number of times and use this for measurement. While this would generate load and allow for measurement, it would not reflect real-world usage, where some functions are called more frequently than others. In this scenario, all functions would be invoked the same number of times, leading to an unrealistic distribution of workload.

Any optimizations that are based on such a measurement would probably have little to no effect for the real users, and there is even the risk of degrading performance by focusing on less critical areas that are not representative of actual usage patterns. For example, if we consider a web application where the login function is called far more frequently than an account settings update function. If we performed the load test with the same n-number of calls to both functions during testing, any optimizations made to the account settings function based on this unrealistic load could end up using resources inefficiently. Some performance optimizations, based on a measurement with this load test, might allocate more resources to the account settings function. This could slow down the login process, which affects most users, while the rarely used account settings function performs marginally better, resulting in a negative impact on performance which the user will suffer from.

So, to create realistic load, the server must be used in the same way, or in a very similar way, as if it was released to the public and used for its intended purpose. One idea is to mimic an end-to-end user interaction with the system. This means that the interactions cover for example creation, manipulation, fetching, and deletion operations but are also idempotent. If they are not idempotent, we would not be able to perform multiple measurements that are comparable because of the different results we would get. Specifically, if some data is requested but then deleted in one measurement, the next measurement would not be able to request this data anymore.

A fast and easy way to mimic an end-to-end user interaction is to record all the user requests that are occurring during the interaction of an actual user with the webpage. Then we can replay those requests to mimic that user. Additionally, we can create multiple copies of the same user to simulate multiple concurrent users at the same time.

Unfortunately, this cannot be applied to every system out there because some systems employ methods against denial of service (DoS) attacks. Due to the fact that this load scenario is executed from one machine and sends a huge amount of requests it comes very close to a DoS attack. Also, there could be measures against the use of bots, by using captchas or similar to make sure the requests come from actual human beings. Developers would need to turn those features off for creating the load test scenarios, which should normally not be a big problem, but leads to less realistic results as the measures also create load on the system which is now missing.

In the following text, we list the tools we have evaluated for our use case. Please note, that we only consider tools that are able to simulate multiple users at once, because a single user alone would not create enough load to yield meaningful results.

**Ranorex**<sup>12</sup> is a tool used at TeamViewer for GUI automation testing. Tested was version 11.3.1. It is primarily designed for functional and regression testing of desktop, web, and mobile applications. While it can be used in automating user interactions with complex interfaces and handling detailed element recognition, it is not well-suited for performing load testing. Unlike tools like Selenium for Python combined with the requests library, Ranorex heavily relies on creating full browser instances or interacting with the desktop environment. This makes it resource-intensive and inefficient for simulating high volumes of concurrent traffic. Additionally, Ranorex's architecture is not designed to handle the scalable, API-focused interactions that are typical in load testing scenarios, where lightweight, parallel requests are essential. Due to these limitations, Ranorex would not be an appropriate choice for our load testing requirements. Also, a commercial license needs to be acquired in order to use it beyond the trial period. We could only find information on the value of the price from third party sites as potential customers need to request a personal quote from Ranorex.

**Selenium**<sup>13</sup> for Python and requests library are packages that can be installed with the built-in package manager in Python. We used Selenium 4.23.0 and Requests 2.32.3. Selenium can create independent browser windows of the most common webbrowsers (e.g. Chrome, Edge, Firefox). It can then be used to navigate a website by searching for the handles (e.g. ID, text, css-selector) of elements on the website and then interact with them. For example we could search for the element "username" and then enter a username into the field. Once it is finished writing we could press the continue button by searching for the text "continue" and calling the method "click" with the handle. The requests library is used to send http requests to the webserver without the overhead of a browser window. This allows for greatly scaling the sending of requests to simulate for example 1000 concurrent users.

In Figure 1 you can see a sequence diagram of the script that creates load, the browser window which is created and controlled by Selenium, and the TeamViewer API which is connected to the webpage. Selenium helps us to automate the login process into our TeamViewer account to create a session that is needed to replay requests. Selenium always creates a new instance of a browser with each run and does not save any data, e.g. no cookies. For a webpage this looks like a first time visit which means that we need to go through a Multi Factor Authorization (MFA) process and need to handle the cookie banner. This was introduced to prevent malicious access to TeamViewer accounts like phishing attacks. TeamViewer implemented two options for MFA to reduce misuse of existing accounts. This can be done either by trusting a device through e-mail or by providing a generated time-based, one-time password (totp). Trusting a device is done by clicking on a link that gets sent to the linked e-mail address. Then again, the confirmation by e-mail would take too much manual effort and time (1-2 minutes for each test run) for automatic testing.

---

<sup>12</sup><https://www.ranorex.com/> last accessed: 14th of October 2024

<sup>13</sup><https://pypi.org/project/selenium/> last accessed: 14th of October 2024

The totp works by receiving a secret key through a QR code, containing a secret code, which can be scanned by an authentication app to calculate a totp. To automate this, we used a python package called pyotp<sup>14</sup>, which takes a secret key and produces a totp, that can be directly used to login.

Once logged in, we transfer all cookies to a session object that was created with the request library. This session can then be used to send http requests as an authenticated user. We can then create multiple threads with this session object to simulate multiple users and send the requests in parallel.

To get the http requests for a certain user interaction we capture the network requests using the built-in Microsoft Edge developer tools (which is based on Google Chrome) and then export them as a har-file (har = http archive)<sup>15</sup>. The har-file is then used to extract the requests with python and saved as a pickle<sup>16</sup> file.

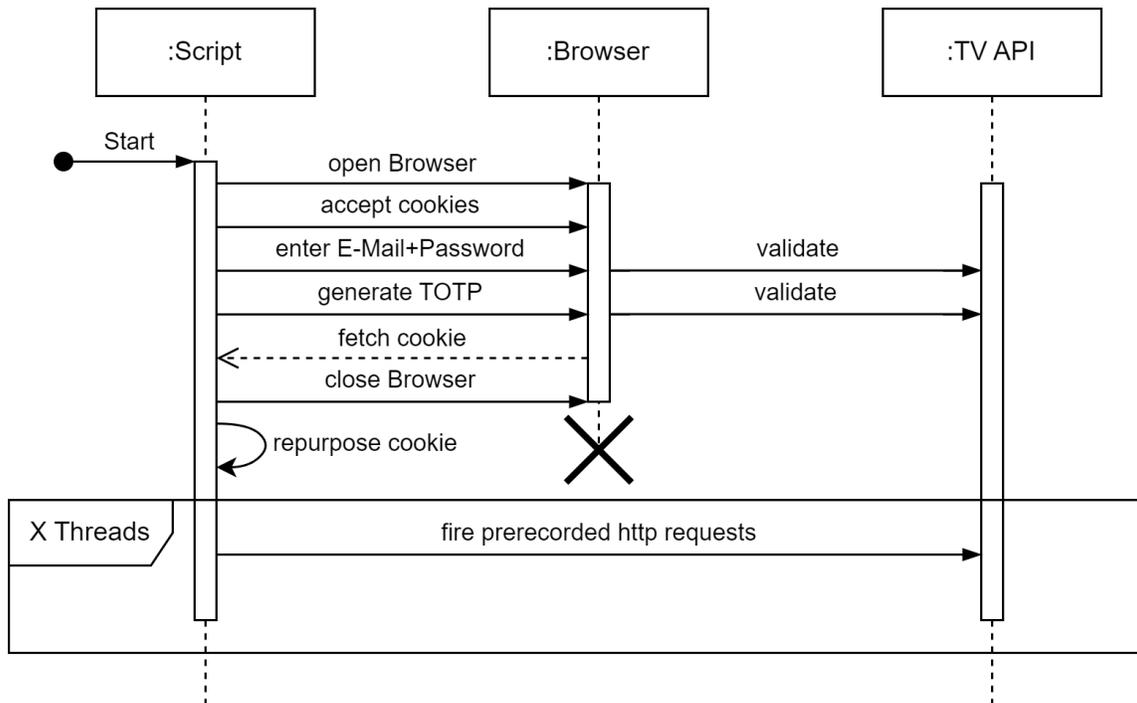


Figure 1: Functionality of the python script to generate load.

**Selenium IDE with Apache JMeter** The Selenium IDE<sup>17</sup> is a browser add-on to record user interactions with the browser. We tested Apache JMeter version 5.6.3 and Selenium IDE 4.0.1-alpha.83. In contrast to the Selenium python library it uses existing cookies if present and stores new cookies and authentication sessions. Thus, it can be used to log in to TeamViewer to be further used by another application. This is a prerequisite for using Apache JMeter, as it is not possible to log in with JMeter itself

<sup>14</sup><https://github.com/pyauth/pyotp> last accessed: 14th of October 2024

<sup>15</sup>A har-file contains data about http requests and their responses in Json format. <https://de.wikipedia.org/wiki/HTTP-Archive> last accessed: 14th of October 2024

<sup>16</sup>Pickle is a Python package to serialize and deserialize objects and can be used to save run-time objects persistently. <https://docs.python.org/3/library/pickle.html> last accessed: 14th of October 2024

<sup>17</sup><https://www.selenium.dev/selenium-ide/> last accessed: 14th of October 2024

because the option to log in via GET and POST request is not available with TeamViewer. Furthermore, due to the storing of cookies and session data, the MFA step does not need to be repeated. So, we only need to set it up once by trusting the device and it will remember this for the following sessions. Apache JMeter<sup>18</sup> is an application to record, construct, and execute load testing scenarios. The amount of concurrent users can be specified and will then be executed by JMeter. However, for load testing it is explicitly recommended that JMeter is run as a CLI interface application as it can happen that the graphical user interface (GUI) freezes and further usage is not possible anymore.

The **approach we chose** for our load test is Selenium for Python with the request library as it is scalable, lightweight, and flexible, because the requests library can send many requests with no overhead of a browser GUI, and is flexible because Python allows to freely design the load test scenario. In addition to that, this solution is open source which allows to adjust the code even further.

## 5.2 First Measurement and Verifying the Measurement Setup

To make sure that the setup is configured correctly and the measurement results are in accordance to research questions **RQ1.3**, **RQ1.4**, and **RQ1.5** we performed some initial measurements. We then compared the measurements for similarity and spread.

Each measurement is performed manually, because due to limitations in the remote desktop connection, sending automated commands via script was not feasible. This means that the load test script is started manually and the profiler is started and stopped manually.

First, the load test is started which performs the login process and then waits for a button press, once it has successfully logged in.

Second, the profiler is started on the VM. Once the message "profiling in progress" is displayed, the load test script resumes and sends requests. The stopping of the load test script after the login is done to reduce the manual error by being able to start sending requests once the profiler starts measuring. In addition to this, the measurement can still be continued if one of the steps fail. For example, the login sometimes did not work due to a timeout. We could then just restart the login script. In contrast, if we started both the profiler and load test without this intermediate step then we would have to stop everything, clean the output and restart everything again for each timeout.

Third, once the load test script was done, the profiler was immediately stopped in the VM. This is also done manually and results in a small measurement difference which needs to be considered.

The first test of the measurement setup consisted of 7 test runs on a working day. The total runtime of the function RequestCombinedCheckResults for each test run can be seen in Figure 2, and is the cumulative time spent in this function during the entire profiling session. The total runtime does not include the time the function spent waiting in any of our results, as this has been filtered out in dotTrace during preparation of the results. The results displayed significant non-linear variability and further investigation is needed to find a reason for this behavior or a solution to reduce this variability.

VM Size	vCPUs	Memory	Network	Temp Disk Size
Standard_A1_v2	1	2 GB	250 Mbps	10 GiB
Standard_D2_v3	2	8 GB	1000 Mbps	50 GiB

Table 3: Two VM size configurations for the measurements.

<sup>18</sup><https://jmeter.apache.org/> last accessed: 14th of October 2024

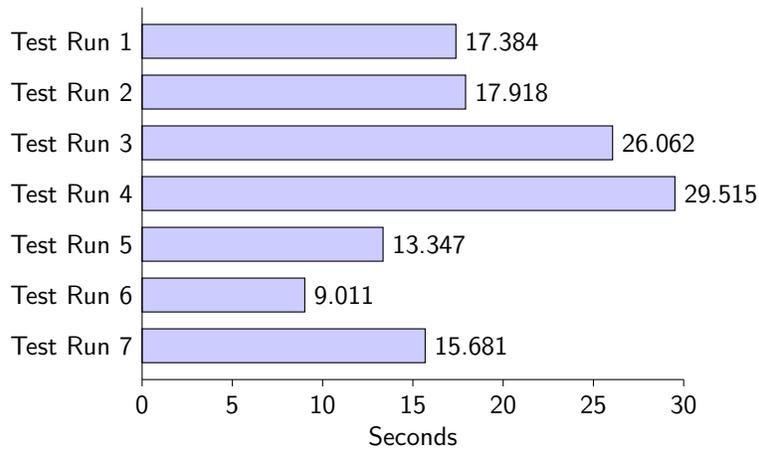


Figure 2: Measurement 1: Total runtime for the function RequestCombinedCheckResults with VM size Standard\_A1.v2.

One reason why the results vary this greatly is that the CPU utilization is too high. When investigating the VM resources, we found that the CPU utilization was close to 100% almost the entire time. The high CPU utilization could lead to the varying results, because of frequent task switching and system interrupts.

To test this, we scaled up the server vertically from Standard\_A1.v2 to Standard\_D2.v3 so that the system does not stay at a CPU utilization of 100% all of the time. For a better understanding of the differences between the two VM sizing options, they are listed in Table 3.

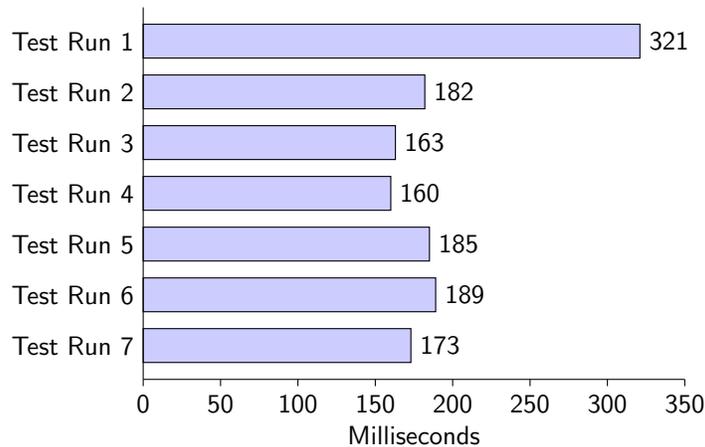


Figure 3: Measurement 2: Total runtime for the function RequestCombinedCheckResults with VM size Standard\_D2.v3 and newer code state.

Changing the VM size led to a change in the mean of the total runtime of the function "Request-CombinedCheckResults" from 18416.86 ms in Figure 2 to 196.14 ms in Figure 3. This decrease in total

runtime of 93.90% can be explained in parts by the upgraded hardware, but hint at another factor that was overseen in the change of measurement setup, as this change is too large for the change we made. One reason could be a performance improvement in the code of the new deployment that leads to the decrease in total runtime, as a newer version of the software was deployed on the VM.

To rule out that this decrease in total runtime originated solely from the VM scaling, we performed another 7 measurements with the VM size `Standard_A1.v2` but deployed the same state of the code. The results for this measurement can be seen in Figure 5 and prove that the scaling was not the only factor that reduced the total runtime.

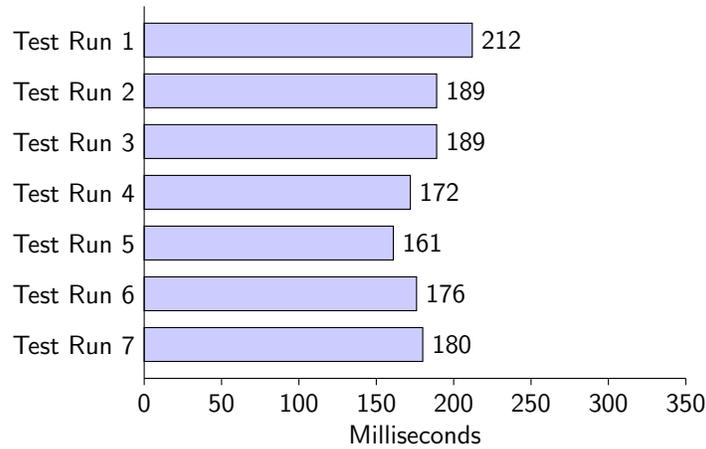


Figure 4: Measurement 3: Total runtime for the function `RequestCombinedCheckResults` with VM size `Standard_D2.v3` on a Sunday and newer code state.

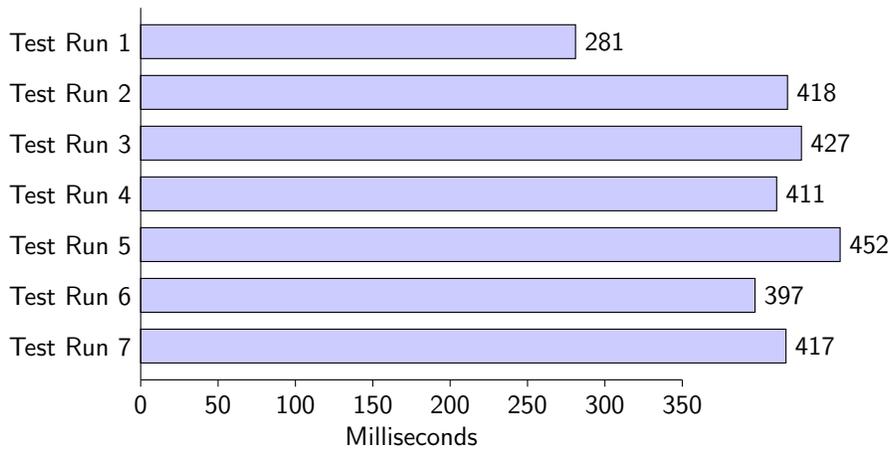


Figure 5: Measurement 4: Total runtime for the function `RequestCombinedCheckResults` with VM size `Standard_A1.v2` and newer code state.

To minimise external noise induced by other users on the system during the measurement we per-

formed 7 measurements on a Sunday afternoon. The results are shown in Figure 4 and compared to the results in Figure 3. To compare them side by side, we created two boxplots which are shown in Figure 6. The two boxplots are quite similar in terms of their central tendencies and overall spread. The median values are 182 for measurement 2 and 180 for measurement 3, indicating that the central points of the data are closely aligned. Additionally, the whiskers of both boxplots, which represent the range of the data, overlap by a great amount, suggesting that the variability within the interquartile range (IQR) is comparable for both groups. However, the key difference lies in the presence of outliers. Measurement 3 has a smaller outlier, while Measurement 2 has a more extreme outlier. These two outliers are also the first in each measurement which could be an indication that there is some kind of caching in place, which allows for faster requests of the same form and data. Another reason could be that the VM falls into a sleeping state and we wake it up with the first load test. This "waking up" of the VM could then be the reason for the higher total runtime.

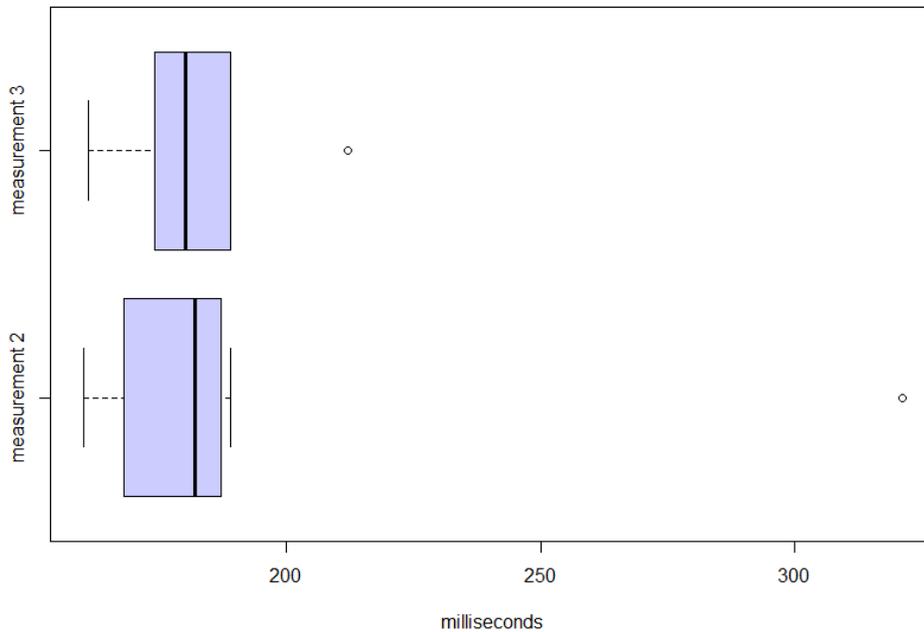


Figure 6: Boxplots for the measurements 2 and 3 of "RequestCombinedCheckResults".

Due to the significantly higher runtimes of the measurement in Figure 2, we do not compare it with the other measurements through a box plot, because the difference is so high that the other boxplots would merely be lines. As a result, we compared measurements 2, 3, and 4 in Figure 7 to show the differences in runtime depending on the VM size. The boxplot of measurement 4 does not overlap with those of measurement 2 and 3. This means that the VM size Standard\_D2\_v3, which measurement 2 and 3 are based on, handles the function significantly faster than the Standard\_A1\_v2 VM size. Consequently, only measurements on the same hardware can be compared without adjusting the results to factor in

the difference in hardware performance.

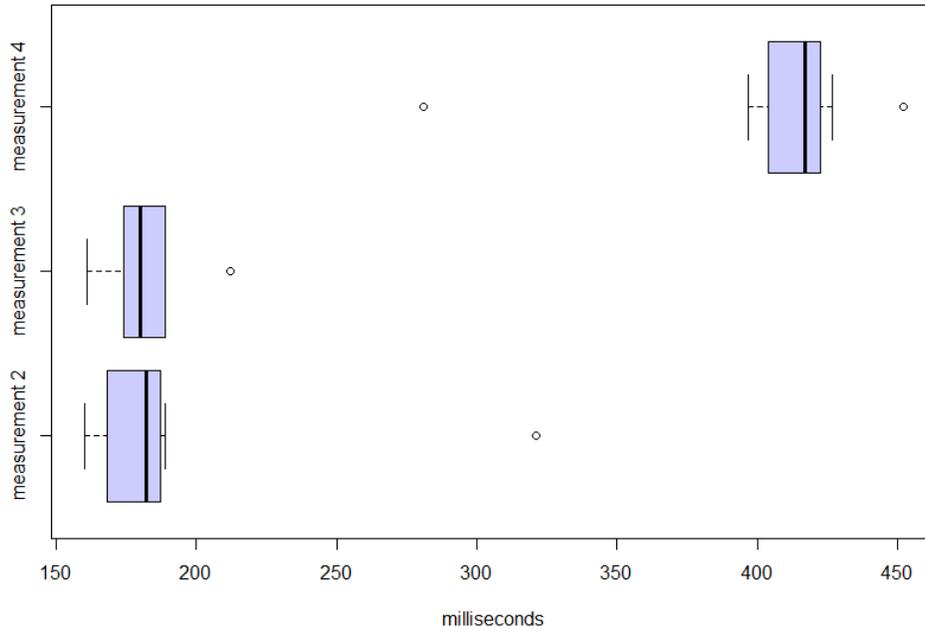


Figure 7: Comparison of the measurements 2, 3, and 4 for the function "RequestCombinedCheckResults". Measurement 4 is performed on the VM size Standard\_A1.v2.

Table 4 summarizes the mean, standard deviation (SD), and relative standard deviation (RSD) of the total runtime of the function "RequestCombinedCheckResults" for each measurement. Measurement 1 has the highest mean value of 18416.86, followed by Measurement 4 with a mean of 400.43, then Measurement 2 with 196.14, and Measurement 3 with the lowest mean value of 182.71.

The standard deviations is not useful, due to the significant difference in means, but are shown anyway for completeness. It was highest for Measurement 1 ( $SD = 6596.44$ ). In contrast, Measurement 3 exhibited the lowest standard deviation ( $SD = 15.00$ ). Measurement 2 and 4 fall between the two with a standard deviation of 51.97 and 51.18 respectively.

To normalize the variability relative to the mean, the relative standard deviation (RSD) was calculated. Measurement 3 had the lowest RSD at 8.21%, indicating the highest precision relative to its mean. In contrast, Measurement 1 showed the highest RSD at 35.82%, highlighting greater relative variability. Measurement 2 exhibited a moderate RSD of 26.50% which includes an outlier. Measurement 4 had an RSD of 12.78%.

Without the outliers, the second measurement would have an RSD of 6.24%, the third measurement would have an RSD of 5.50%, and the fourth measurement would have an RSD of 4.00%. The first measurement does not have outliers.

During the first two measurements we identified key differences and made adjustments to scale

Measurement	VM size	Mean	SD	RSD
1	Standard_A1_v2	18416.86	6596.44	35.82
2	Standard_D2_v3	196.14	51.97	26.50
3	Standard_D2_v3	182.71	15.00	8.21
4	Standard_A1_v2	400.43	51.18	12.78

Table 4: Statistic differences of measurements per VM configuration for the function RequestCombinedCheckResults.

up the VM size. After adapting the VM size, the data seemed to demonstrate consistency across all measurements. On the other hand, after deploying the code state used in measurement 2 and 3 to a VM with the same size as in measurement 1 we also got consistent results. The noise induced by other users on this particular system during the workday is negligible or even nonexistent and further measurements can be performed during this time.

The reduced variability and alignment of the statistical indicators suggest that the measurements are reliable, and the results are reproducible. This confirms that the methodology is robust and capable of providing dependable outcomes for other measurements.

### 5.3 Measurement of degrading performance

In this section we simulate a performance degradation which would also result in a higher energy consumption due to the additional processing needed. Performance degradation can occur during the development process in many different ways. It could happen through the change of libraries, data types, and redundancy to name just a few. To simulate our performance degradation we call a function twice instead of once within the function "RequestCombinedCheckResults". The redundant function parses the entire data that is returned with a runtime complexity of  $\mathcal{O}(n)$ .

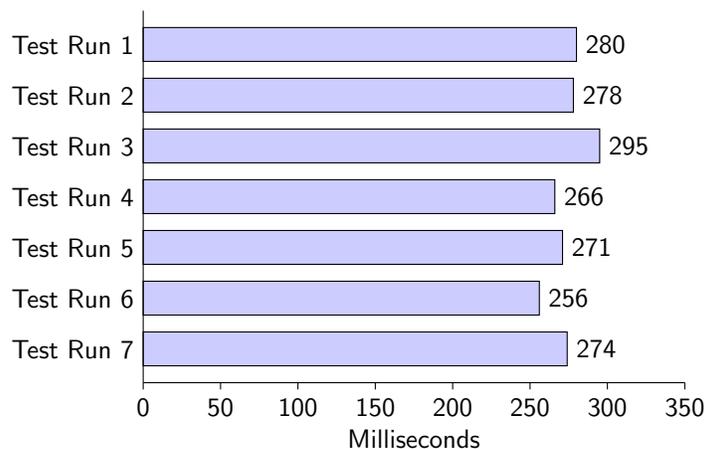


Figure 8: Measurement 5: Total runtime for the function RequestCombinedCheckResults with VM size Standard\_D2\_v3 with performance degradation.

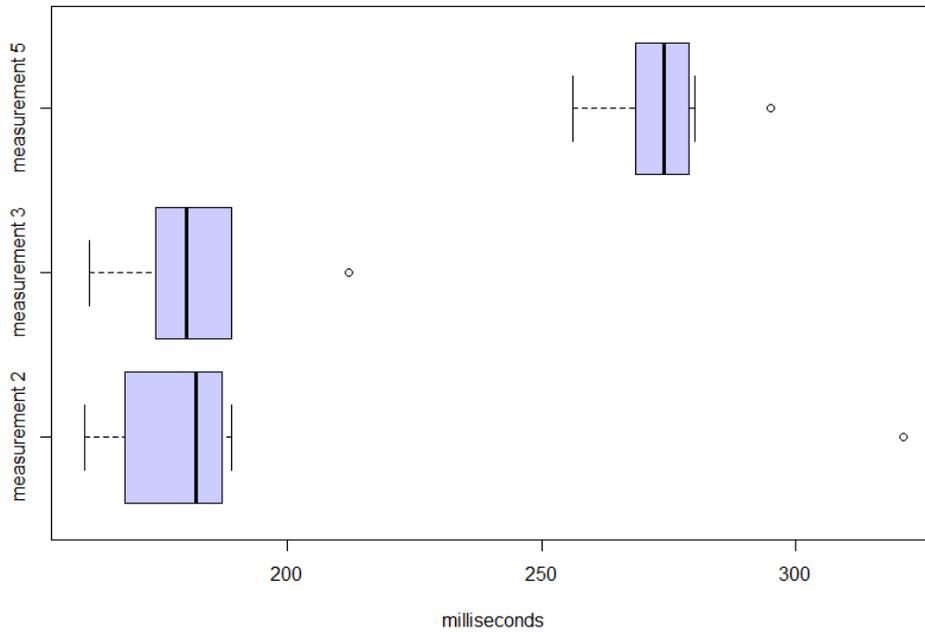


Figure 9: Comparison of the measurements 2, 3, and 5 for the function "RequestCombinedCheckResults". Measurement 5 shows the execution time for a performance degradation.

In Figure 8 the test runs of measurement 5 are presented to give a visual overview of the individual results. Similar to the other measurements, measurement 5 consisted of 7 test runs and it has a mean of 274.29 ms, a standard deviation of 11.27 and a relative standard deviation of 4.11%. This code change led to a runtime increase of 84.86 ms when comparing the mean of measurement 5 to the combined mean of measurements 2 and 3. This can also be seen in Figure 9 where the boxplots only overlap with one outlier.

The measurement shows that the runtime of the function increased by 44.8% aligning with the expected performance degradation. This performance degradation also implies a reduced energy efficiency due to the extra computation time the CPU needs to handle the function.

## 6 Discussion

In our experiment with intentional performance degradation, CPU-time increased as expected in measurement 5 which can be seen in Figure 9. The duplication of a function call within the "RequestCombinedCheckResults" function resulted in a substantial increase in total runtime of the function by 44.8%. For the constraints mentioned earlier, namely measuring inside of a VM, we assumed that the CPU-time is a metric that correlates closely enough to make statements about the energy consumption. With this assumption, our experiment proves its ability to spot code changes that affect the energy consumption.

The measurements could be subject to several potential biases and errors due to the nature of the test setup. First, the manual start and stop of both the load test and the measurements lead to time variations. Inconsistent start and stop times could lead to inaccuracies in the measurement window and affect the accuracy of the recorded CPU time. In addition, human reaction time can vary when stopping the measurement, which can result in the actual CPU utilization for the tasks being profiled being recorded too high or too low.

Research question **RQ1.6** is about verifying the correlation between the measurement results and the absolute energy consumption of the software. We measure CPU time, but do not directly measure energy consumption. Therefore, assuming that CPU utilization directly correlates with energy consumption may lead to errors. The relationship between CPU time and energy consumption may be influenced by factors such as CPU frequency scaling, energy management features, and hardware efficiency, which were not directly controlled or measured in this experiment.

Another possible source of error is the fluctuating system load during the tests. As both the measurement and the load test were carried out manually, background processes or resource conflicts may have occurred in the system, which could distort the CPU utilization data. These uncontrolled factors could lead to an under- or overestimation of energy consumption based on the current CPU load, which is not directly attributable to the measured functions.

From time to time the VM's are restarted with different hardware which leads to slightly different measurements. Although the VM size specifies the hardware to some extent, there is still a difference between the CPU models used within a VM size. This is expected as it is hard to always provide the same CPU model for a large amount of servers and over the course of years. For example, the change to a newer CPU model is not done for all servers at once but gradually.

We assume a computation heavy workload in which the CPU causes the largest part of the energy consumption. For workloads where other parts of the system have a higher energy consumption, the results of the measurements are skewed. For example, energy consumption differences caused by workloads with many read and write operations on the storage device will not be shown adequately in the measurement results.

Only one level of load has been introduced during load testing instead of multiple levels with varying intensity. For example, light, moderate, and heavy usage scenarios could be created to better reflect the range of usage that real-world applications typically encounter. The difference in load could have effects on the system that we could not yet see and lead to a more realistic mean value as the load in a real system can also vary greatly during operation.

## 7 Conclusion and Future Work

The goal of this thesis was to develop a method to measure energy efficiency of software within a VM that can be used in a CICD pipeline. The thesis provides a method for continuous monitoring of energy efficiency at the function level, between two code states. The use of profiling tools allowed for detailed measurements of CPU usage, and software inefficiencies could be detected, allowing for targeted optimizations aimed at reducing energy consumption.

The approach in this work can be used for detecting performance degradation in software systems over time. By profiling CPU utilization over time, deviations in function execution times can be identified which allows to tackle the performance painpoints directly. Due to the correlation between function runtime and energy efficiency, this also helps with reducing the energy consumption.

The approach we used enables drilling down into every executed function. This could be used to measure test coverage in a future project by analyzing which functions have been executed. It is important to use an instrumentation profiler in this case as a sampling profiler might miss parts of the code.

The experiment was conducted on a single system under specific conditions, which may limit the applicability of the results to other hardware configurations or workloads. Hence, to improve this in the future, we could use a more controlled environment and possibly a larger sample of systems for a broader analysis.

To enhance the accuracy and reliability of the measurements, future work should verify the correlation between CPU time inside a function and energy consumption through direct energy measurement tools. This would offer more confidence in the correlation between measured CPU activity and actual energy consumption.

Additionally, cross-validation of CPU utilization data from known benchmarks could be used to compare measurements between different hardware configurations. For example, a standardized CPU benchmark score of 1000 points for a system with CPU A and a score of 700 points for a system with CPU B could be used to compare measurements on the two systems by factoring in the relative difference between the two scores for the measurements.

Finally, automating the integration of these measurements within CICD pipelines could ensure that software energy efficiency is tracked over time. This would allow for ongoing monitoring and optimization as software evolves, ensuring that efficiency remains a priority in future software development processes. Extending on that idea, for any performance degradation there could be an alarm that notifies the corresponding developers which function performed badly and compare the current codes state to the former.

## References

- [1] Beth Whitehead et al. "Assessing the environmental impact of data centres part 1: Background, energy use and metrics". In: *Building and Environment* 82 (2014), pp. 151–159. ISSN: 0360-1323. DOI: 10.1016/j.buildenv.2014.08.021 (cit. on p. 1).
- [2] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. "Data center energy consumption modeling: A survey". In: *IEEE Communications surveys & tutorials*. Vol. 18. 1. IEEE, 2015, pp. 732–794. DOI: 10.1109/COMST.2015.2481183 (cit. on p. 1).
- [3] Luca Ardito et al. "Methodological guidelines for measuring energy consumption of software applications". In: *Scientific Programming* 2019 (2019), pp. 1–16. DOI: 10.1155/2019/5284645 (cit. on p. 1).
- [4] Alexandru G Bardas et al. "Static code analysis". In: *Journal of Information Systems & Operations Management* 4.2 (2010), pp. 99–107 (cit. on p. 4).
- [5] NH White and Keith H. Bennett. "Run-time diagnostics in pascal". In: *Software: Practice and Experience* 15.4 (1985), pp. 359–367. DOI: 10.1002/spe.4380150405 (cit. on p. 4).
- [6] Wentai Wu et al. "A Power Consumption Model for Cloud Servers Based on Elman Neural Network". In: *IEEE Transactions on Cloud Computing* 9.4 (2021), pp. 1268–1277. DOI: 10.1109/TCC.2019.2922379 (cit. on pp. 5, 6).
- [7] Hao Zhu et al. "Estimating Power Consumption of Servers Using Gaussian Mixture Model". In: *2017 Fifth International Symposium on Computing and Networking (CANDAR)*. 2017, pp. 427–433. DOI: 10.1109/CANDAR.2017.44 (cit. on pp. 5, 6).
- [8] Norbert Schmitt et al. "Online Power Consumption Estimation for Functions in Cloud Applications". In: *2019 IEEE International Conference on Autonomic Computing (ICAC)*. 2019, pp. 63–72. DOI: 10.1109/ICAC.2019.00018 (cit. on pp. 5–8).
- [9] Yanfei Li et al. "An Online Power Metering Model for Cloud Environment". In: *2012 IEEE 11th International Symposium on Network Computing and Applications*. 2012, pp. 175–180. DOI: 10.1109/NCA.2012.10 (cit. on pp. 5, 7).
- [10] Gaurav Dhiman, Kresimir Mihic, and Tajana Rosing. "A system for online power prediction in virtualized environments using Gaussian mixture models". In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California: Association for Computing Machinery, 2010, pp. 807–812. ISBN: 9781450300025. DOI: 10.1145/1837274.1837478 (cit. on pp. 5, 7).
- [11] Aman Kansal et al. "Virtual machine power metering and provisioning". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 39–50. ISBN: 9781450300360. DOI: 10.1145/1807128.1807136 (cit. on pp. 5, 7, 8).
- [12] Deguang Li et al. "Software Energy Consumption Estimation at Architecture-Level". In: *2016 13th International Conference on Embedded Software and Systems (ICCESS)*. 2016, pp. 7–11. DOI: 10.1109/ICCESS.2016.35 (cit. on p. 6).
- [13] Fadi Wedyan, Rachael Morrison, and Osama Sam Abuomar. "Integration and Unit Testing of Software Energy Consumption". In: *2023 Tenth International Conference on Software Defined Systems (SDS)*. 2023, pp. 60–64. DOI: 10.1109/SDS59856.2023.10329262 (cit. on p. 6).

- [14] Dirk Beyer and Philipp Wendler. “CPU Energy Meter: A Tool for Energy-Aware Algorithms Engineering”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Armin Biere and David Parker. Cham: Springer International Publishing, 2020, pp. 126–133. ISBN: 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7\_8 (cit. on p. 8).

Sources were last checked on 9th of October 2024.