TU WIEN Informatics

CoMECS: Framework for Comparing and Measuring the Energy Consumption of Software

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Patric Gruber, BSc
Matrikelnummer 11775839

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Univ.-Prof. DI Dr.techn. Johanna Ullrich, BSc

Wien, 1. September 2025

Patric Gruber	Edgar Weippl

TU WIEN Informatics

CoMECS: Framework for Comparing and Measuring the Energy Consumption of Software

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Patric Gruber, BSc
Registration Number 11775839

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Univ.-Prof. DI Dr.techn. Johanna Ullrich, BSc

Vienna, September 1, 2025

Patric Gruber	Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Patric Gruber, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 1. September 2025

Patric Gruber

Kurzfassung

Der menschen-gemachte Klimawandel ist eine der größten Herausforderungen der früheren, momentanen und kommenden Generationen. Menschen produzieren eine riesige Menge an Treibhausgasen, welche die Atmosphäre erhitzen. Einer der Hauptgründe hierfür ist das immense Verlangen nach Energie, das wir als Gesellschaft haben. Der Bericht *Statistical Review of The World* von 2024 zeigt, dass die Welt nur ungefähr 20% der Gesamtenergie aus erneuerbaren Quellen gewinnt. Der Rest produziert Treibhausgase. Deshalb müssen wir den vollständigen Übergang zu erneuerbaren Energieträgern schaffen um diese Emissionen zu verhindern. Das ist jedoch nicht schnell genug möglich um den Klimawandel aufzuhalten, weshalb neben der Transformation der Energieproduktion, auch eine Reduktion des globalen Energieverbrauchs nötig ist. Das inkludiert jeden Aspekt der Wirtschaft, was natürlich auch für die IT-Branche eine Evaluierung und Reduktion ihrer Auswirkung zur Folge haben muss. Malmodin und Lundén schätzen in ihrer Publikation von 2018, dass der Energiebedarf der Bereiche ICT und Entertainment & Media in Zukunft fallen werden. Für Datenzentren hingegen wird geschätzt dass deren Bedarf steigen wird. Der neueste Bericht der International Energy Agency von 2025 schätzt, dass sich der Energieverbrauch von Datenzentren bis 2030 mehr als verdoppeln wird. Dies ist vorallem der Popularität von Anwendungen im Bereich der Künstlichen Intelligenz zu verdanken.

Der IT-Sektor, und im Speziellen der Teil der sich auf Serverinfrastruktur fokussiert ist deshalb besonders gefragt, den eigenen Energieverbrauch zu messen und zu reduzieren. Ein großer Teil moderner Serverinfrastruktur baut auf Open-Source Software auf, welche meistens von einem kleinen Anzahl von Freiwilligen gewartet und entwickelt wird. Diese haben nicht die Resourcen um aufwendige Messungen des Energieverbrauchs ihrer Software durchzuführen. Dies wird zusätzlich durch die fehlende Auswahl an frei verfügbaren und gut getesteten Programmen zur Stromverbrauchsmessung erschwert. Weiters sind die verfügbaren Programme sehr unterschiedliche in dem Format der Dateneingabe, -ausgabe und deren Verwendung.

Deshalb stellen wir ein Framework und Methodik vor die es ermöglicht den Stromverbrauch von Software mit einer großen Anzahl unterschiedlicher software-basierter Strommesser zu messen. Die drei Ziele die wir hierfür definieren sind *Offenheit*, *Reprouzierbarkeit* und eine *Standardisierte Ausgabe*, welche das von uns implementierte Framework *CoMECS* erfüllen muss. Wir zeigen wie eine Wrapper-Struktur die auf Nix

aufbaut, reproduzierbare Experimente erlaubt. Zusätzlich präsentieren wir unseren Ansatz nur mithilfe von SSH einen Computer zu steuern, auf welchem der Stromverbrauch gemessen wird. Diese Techniken erlauben es zuverlässige und reproduzierbare Tests zu definieren und auszuführen. Diese können dann auf einem Desktop Computer oder einem Server ausgeführt werden ohne Veränderung der vorher definierten Tests.

Wir verwenden unser Framework um sechs software-basierte Strommesser mit einer hardware-basierten Lösung zu vergleichen. Die Ergebnisse zeigen, dass die Werte von der `pkg`-Domäne des RAPL-Sensors und das *CodeCarbon* Tool am ähnlichsten zu den Messwerten des Hardware-Geräts sind. Dies ist allerdings nur bei CPU- und Hauptspeicherfokussierten Lasten der Fall. Ein anderes von uns durchgeführtes Experiment reproduziert die Ergebnisse einer früheren Publikation zum erhöhten Energieverbrauch der durch Containerisierung mittels Docker entsteht. Schließlich evalieren wir den Stromverbrauch von nginx über 16 Versionen hinweg und zeigen, dass der durchschnittliche Stromverbrauch zwischen den Versionen *1.11* und *1.26* um ungefähr 3% abnimmt.

Abstract

The current human-made climate change is one of the greatest challenges of the last, current and coming generations. Humans produce a huge amount of greenhouse gases, that warm up the atmosphere. The immense thirst for energy that we as a society have is one of the main drivers. Achieving the ambitious goals set in the agreements that most countries of the world signed, will necessarily mean reducing our global energy consumption. The *Statistical Review of The World* report from 2024 shows that the world only produces around 20% of its energy from renewable sources. The rest emits greenhouse gases during its production. Thus we have to transition to renewable energy to avoid these emissions. This is however not possible to do quickly enough, so besides transforming the energy production, we also have to reduce our energy consumption globally. This includes every single part of the economy, which of course means that IT also needs to evaluate and reduce its impact. Malmodin and Lundén estimate in their paper from 2018 that the energy consumption of the ICT and Entertainment & Media sector will fall overall in the future, but for data centers it will rise. The newest report from the International Energy Agency from 2025 estimates that the energy consumption of data centers will more than double by 2030, especially due to the rise in popularity of AI applications.

The IT sector and specifically the part of it that is focused on server infrastructure therefore has to get ready to measure and reduce its energy consumption. Underlying a huge part of modern server infrastructure is open-source software, which is usually maintained by just a small number of volunteers. These people do not have the resources to do extensive testing of the energy consumption of their software. This problem is amplified by the lack of well tested and freely available tooling that can be used to measure the power usage of programs. Additionally the available tools are very different in their input, output and their usage.

We therefore propose a framework and methodology which facilitate the measurement of energy consumption of software with a wide variety of available software-based power meters. The three goals that we define for this are *Openness*, *Reproducibility* and *Standardized Output*, which our framework *CoMECS* has to fulfill. We show how we built a wrapper structure based on Nix, which allows for reproducible experiments. We also present our approach to using SSH for controlling the test machine, on which the energy consumption is measured. These techniques allow us to define and execute reliable and

reproducible tests against a desktop machine and server without changing any of the defined wrappers.

We then use our framework to compare six software-based power meters against a hardware-based solution. Our results show that using the values from the `pkg` domain of RAPL sensor or the *CodeCarbon* tool most closely resembles the real measurements using a hardware-based power meter. This is however only true for CPU and main memory focused work loads. Another experiment we conducted was reproducing previous results of how much energy consumption overhead containerization using Docker causes. Lastly, we evaluated the power consumption of nginx over the course of 16 releases and found that it reduced its average energy consumption between the version *1.11* and *1.26* by around 3%.

Contents

Kurzfassung vii

Abstract ix

Contents xi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research Questions . . . 3
- 1.3 Methodology . . . 4
- 1.4 Structure . . . 5

2 Background 7
- 2.1 Electricity . . . 7
- 2.2 Measurement with Hardware Meter . . . 8
- 2.3 Running Average Power Limit (RAPL) . . . 10
- 2.4 Model-based Estimations . . . 12
- 2.5 System-wide vs Application-specific . . . 13
- 2.6 Related Work . . . 13

3 Method 17
- 3.1 Goals . . . 17
- 3.2 Overview . . . 18
- 3.3 Details . . . 22
- 3.4 Discussion . . . 33
- 3.5 Conclusion . . . 35

4 Evaluation & Results 37
- 4.1 Power Meters . . . 37
- 4.2 Docker: PostgreSQL . . . 41
- 4.3 Parameterization: nginx . . . 44
- 4.4 Artificial Intelligence and GPUs . . . 48
- 4.5 Conclusion & Future Research . . . 52

5 Conclusion **55**

6 Appendix **59**
6.1 Wrapper Examples 59
6.2 nginx Configuration 61
6.3 Configuration files JSON schema 62

Overview of Generative AI Tools Used **67**

Übersicht verwendeter Hilfsmittel **69**

List of Figures **71**

Bibliography **73**

CHAPTER 1

Introduction

1.1 Motivation

In the 19th century scientists discovered the greenhouse effect causing a temperature increase due to greenhouse gases trapped in the atmosphere. This effect was experimentally verified by Eunice Newton Foote in 1856 [24]. Since then, it could be observed in the atmosphere of our planet as greenhouse gas (GHG) emissions have increased massively since the industrial revolution. Average global temperature has risen ever since. Today, there is a vast body of scientific literature modeling this effect and predicting how this influences weather trends over time (called climate), extreme weather phenomena, pandemics and much more. This trend, called climate change, has become the major challenge of our time, which can be seen by the effort put into the topic by the scientific community and how widely it is discussed in the public, politics and the news.

In 1992, the United Nations formed the *United Nations Framework Convention on Climate Change* (UNFCCC) during the *Rio Conventions* [56], with the goal to support the international community in fighting human-made climate change [59]. In 2005, the *Kyoto Protocol* was signed by 192 industrialized nations committing themselves to reduce their GHG emissions significantly [57]. In 2015, the *Paris Agreement* was signed by the members of the *UNFCCC*. This proposal stated the goal of limiting the average temperature rise to below 2°C compared to pre-industrial levels and to keep it to 1.5°C [58]. Since signing the *Paris Agreement* it became increasingly clear that 1.5°C of average temperature rise should be the target to aim for, since anything more than that will have profound and lasting impacts on humanity. If the world wants to achieve this, the global GHG emissions have to peak before 2025 and be reduced by 43% in 2030 according to the Intergovernmental Panel on Climate Change (IPCC) [23]. Additionally to this agreement, the European Union has implemented *The European Green Deal* [11] which sets the goal of becoming net zero by 2055 and reduce the GHG emissions by 55% till 2030.

These targets can not be achieved by minor adjustments. The climate crisis is the ultimate cross-cutting concern, because every part of human life including agriculture, industry, transportation and energy needs to be adjusted accordingly. Obviously, this also includes software engineering, which is deeply intertwined with today's society. We communicate via instant messaging. We buy our groceries and food via apps and delivery services. We use smart phones, tablets, laptops and desktop PCs for entertainment and work. Software is everywhere and its GHG emissions need to be reduced, like it has to in every other part of society.

Malmodin and Lundén looked at the energy and carbon footprint of the ICT and entertainment & media (E&M) sectors from 2010 to 2015 [28] and found a steady decline within this period. They also predicted that the emissions of these industries will fall further until 2020. This is however only true for the ICT and E&M sector as a whole. The energy consumption for data centers grew steadily. Not exponentially, as the number of consumers/users, but it was still increasing instead of declining.

In the 2025 report titled *Energy and AI* [20] that was published by the International Energy Agency, they estimated that in 2024 the amount of energy used by data centers alone was 415TWh, which amounts to around 1.5% of the global electricity consumption. The report also projects the amount of electrical energy used by data centers in 2030 could rise to 945TWh. This is mainly due to artificial intelligence (AI) being a main focus of investment right now and these applications being extremely resource intensive.

Data centers therefore need to become much more energy efficient to achieve a decrease with the projected growth of users and energy intensive tasks such as AI. Using power necessarily means that carbon emissions are created as long as the generation of electricity is coupled to emitting greenhouse gases. It is therefore necessary to reduce the energy consumption as far as possible, so that the demand can be satisfied with the limited amount of carbon-neutral renewable energy that humanity is capable of producing.

Before being able to reduce it, a metric that measures the carbon footprint of software is needed. Otherwise there would be no possibility to even quantify it and therefore no possibility to track changes. One such way is the *Software Carbon Intensity* (SCI) defined by the Green Software Foundation [14]. This metric makes it possible to compare different programs or versions of the same software regarding their power consumption. The base formula of the SCI is

$$\text{SCI} = ((E * I) + M)/R$$

E is the used energy, I is the carbon intensity, M are the embodied emissions and R functional unit.

It basically describes the amount of produced emissions per action, i.e. adding one user to the application, executing this task, etc. For this the energy is needed that is produced by the action. This can be either estimated or measured. Since estimation is easier than actually measuring energy consumption, smaller companies or open source software usually choose this option, rather than measuring it accurately, in case they are

even tracking their carbon footprint at all. This however can cause problems, since the estimation is always just that, an estimate.

Large IT companies that use a lot of energy for their data centers also pay a lot of money for it. So they are eager to make their in-house server software more efficient to save money. For open source software that does not really apply, since there is rarely a single company behind it, that runs this software in huge data centers and therefore has to bear the costs of inefficiencies. Open-source software is predominantly used by a lot of different people and companies and therefore not a single entity bears the operational costs. Yet, the aggregated energy consumption is still huge, which has the potential to significantly reduce carbon emissions by making it more efficient.

If there would be a tool that allows to easily measure the energy consumption, the SCI could be calculated precisely for open-source and other software and engineers could define targets and tests for the energy consumption of their software. This could help the IT industry to contribute to the goal of limiting the global warming to 1.5°C.

Creating such a tool is the goal of this thesis.

1.2 Research Questions

The following research questions are the basis of this thesis.

RQ1: *How can energy consumption be measured? What are the differences between different approaches?*
It is important to find tools and techniques that can measure or estimate the energy consumption of software and to analyze them regarding their usage, interface and output. The answer reveals the state-of-the-art, the similarities and differences of the techniques. It allows to define an useful evaluation to compare these tools. The results of that comparison shows if the accuracy of software solutions are comparable to that of dedicated hardware. Based on the result it can be stated which technique is the best to use in which context.

RQ2: *What are the requirements for a measurement framework for software allowing different measurement techniques?*
Evaluating the requirements needed for a framework that wraps the different tools allows to implement such a framework and then to actually evaluate the measuring techniques effectively.

RQ3: *How can a unifying framework be used for real software? What advantages does this approach have?*
The results of this step show why such a framework is useful beyond just comparing different measuring techniques. It also allows to evaluate the possibilities for future research and what the real world impact is for software development in general.

Answering all these questions allows to present a feasible solution to measure the carbon footprint of software without relying on rough estimates. This allows to define specific and measurable targets to reduce the carbon footprint of widely used software, e.g. open source web servers or databases, which can have a significant impact on the whole industry, since these types of software are widely used and even small changes sum up to a lot of savings.

As an example of how much impact this can have, we will use *nginx* [12]. It is an open-source web server and is market leader according to different sources ([60, 7, 31, 50]). It is estimated to have a market share of around 30%. Based on an estimated 200 million total active websites [22, 51] we can assume that 60 million websites are served by *nginx*. This means that reducing the energy consumption even slightly would have a tremendous cumulative effect.

1.3 Methodology

Answering the research questions defined in section 1.2 is the goal for this thesis. Achieving this is done by following the three main steps, which are named *Literature Review*, *Framework Goals* and *Implementation and Evaluation*.

Literature Review In this step we present the findings of the literature review, that shows which approaches and tools are currently available to researchers. Each tool is analyzed regarding its usage and output. We present six software-based power meters that are freely available to researches at the time of writing. We show that they are quite different in their handling and explain how this is a challenge when comparing them.

Framework Goals The definition of the goals for the framework are essential for effectively implementing it and being able to evaluate if it answers the research questions. The goals include *Openness*, *Reproducibility* and *Standardized Output*. When these are properly implemented, they result in a general framework that can be used for different types of experiments that help researchers to efficiently test their hypothesis and allow others to reproduce their results.

Implementation After establishing the goals for the framework and the challenges presented by the heterogeneous nature of the available tools, it can be implemented. We present the challenges that needed to be overcome before being able to remotely control the test machine in a reliable fashion and what decisions were made in the implementation process that lead to a framework that fulfills all the requirements.

Evaluation The resulting program is then evaluated against the defined goals. This is done in three experiments. The first one serves the purpose of comparing the tools identified in the literature review between themselves and against a hardware-based solution from a previous master thesis [8]. The second experiment shows how previous

research regarding the impact of containerization on the energy consumption of software, can be reproduced using the implemented framework. The last experiment shows that the tool allows to define and executed complex experiments on real world software.

1.4 Structure

This thesis is structured into four chapters.

Chapter 2 describes the basics of electronics, that are necessary to understand how energy consumption can be measured. This includes voltage and current, units and their conversions and what energy consumption is. Additionally the different approaches of measuring it are explained. After that the methods of energy consumption measurements for software are described. These include hardware based solutions that use an external tool, software based approaches, like perf [42], PowerJoular [35], and hybrid approaches, like the RAPL sensor [21]. The different techniques are compared and their respective advantages and disadvantages are discussed. Additionally, the available state-of-the-art open-source tools and their properties are described, which include their output format, output units and what capabilities they offer.

Chapter 3 presents the considerations and steps that were taken when designing and implementing a framework for unifying the different measurement approaches. First, a high-level overview is shown, which includes the most important parts and their interaction. Then, the most interesting components are analyzed and presented in detail. These are mostly the interaction between the test runner and the tested machine, how dependencies are handled and how different software is packaged into wrappers that CoMECS can use. After that the wrapper approach for meters and tests is discussed, we argue how this facilitates the reproduction of the results on different hardware with minimal effort. Then we describe the parameterization of wrappers and how this allows to define a series of tests, e.g. multiple different versions of the same software, very easily with just a single wrapper. Lastly, the overall test process is presented. This explains why there is a baseline and system test, before the actual test runs are executed and how the final results are calculated from these measurements.

Chapter 4 describes how the research questions are translated into specific tests and experiments which answer that question. It explains how the tests were designed to facilitate the comparison of the different meters on server and desktop hardware. Additionally, it shows how previous research has been validated regarding the overhead of containerization and it also discusses how the energy consumption of the web server *nginx* changed since version *1.11*. Additionally, it debates how CoMECS could be used in future research and in what directions the framework could be developed further.

Chapter 5 summarizes the whole thesis and concludes it. This chapter picks up the most important points about the motivation for reducing the carbon emissions and why the IT sector has to contribute to the fight against climate change. The goals we set for our framework and the steps to implement them into a concrete framework are briefly

discussed. Then the experimental setup and their results are presented and a future outlook of possible applications of the framework or methods discussed in this thesis is given.

CHAPTER 2

Background

This chapter discusses the basics of how energy consumption can be measured. First, we discuss the underlying basics of electricity and what "energy consumption" is. Then, the three approaches of measuring are compared, i.e. hardware power meter, the RAPL sensor and software power meter. Last, the current state of the art is presented, which includes the current approaches and tools for energy consumption measurements of software.

2.1 Electricity

The basic building blocks of electricity are voltage, current and resistance. Their relation is usually presented in form of Ohm's Law

$$U = R * I$$

U is the *Voltage* (measured in Volt V), R is the *Resistance* (measured in Ohm Ω) and I is the *Current* (measured in Ampere A).

Power is defined as the rate of work. The SI unit is Watt W and the formula is

$$P = U * I$$

Energy is the amount of work done. Commonly kilo-watt hours (kWh) are used for this unit, but Joules J is the actual SI unit. The formula is

$$E = P * t$$

The relation between J and kWh is defined as

$$1kWh = 3.6MJ$$
$$or$$
$$1Ws = 1J$$

Example: When a PC with components that have a total of $300W$ of (steady) power usage, it means that per hour of running time the PC uses up $300Wh$ of energy, which is equal to $1.08MJ$.

The energy is the unit we are interested in. Even though energy can not be lost or destroyed (because of the first law of thermodynamics), it is converted from usable electric energy into unusable one (e.g. heat).

This conversion is what we mean by the term *energy usage*.

2.2 Measurement with Hardware Meter

Electrical power can be measured with sensors. There are sensors that are integrated into the load bearing circuit and others that are "contact-less", meaning that they are not part of the actual circuit, but measure the effects of the electricity instead [8]. One such class of sensors use the Hall effect, which is the name for the magnetic field that is induced by the electricity flowing through a wire. The disadvantage of the contact-less sensors is that they are more expensive and complex to correctly use.

The hardware measurement device that serves as the baseline for all the experiments that have been conducted for this thesis, is a device built for the master thesis of Sascha Böck [8]. The power meter uses the energy meter module ADE9153A [3]. It is a fully integrated solution, which is part of the circuit itself. The sensor has a resolution of 4kHz and allows to measure power consumption of electricity with a voltage of 240V/AC and currents of 5-10A.

Figure 2.1 shows how the setup for using the hardware meter works. It provides two ports, which are a power plug and a USB port. The device for which the power consumption is measured is connected to the power plug, which provides the electricity to the measured device. The USB connection provides a serial port which is used to control the power meter and to retrieve the data. The state machine of the power meter and how to control it is shown in figure 2.2. Before being able to measure the power consumption the sensor needs to be calibrated, which is done by sending the character `c`. This takes 60 seconds, after which the meter first sends the calibrated parameters and then immediately starts to measure and send the data via the serial port continuously. The data is packed as binary data in a C-struct format, which is shown in figure 2.3. After the experiment is done, the character `q` is send to stop the hardware meter from sending any more data and return to the *Waiting* state. The meter sends 4000 data packets of the type shown in figure 2.3 per second. They have to be unpacked before being able to use them. In Python this can be done with `struct.unpack('fffff', rawData)`.

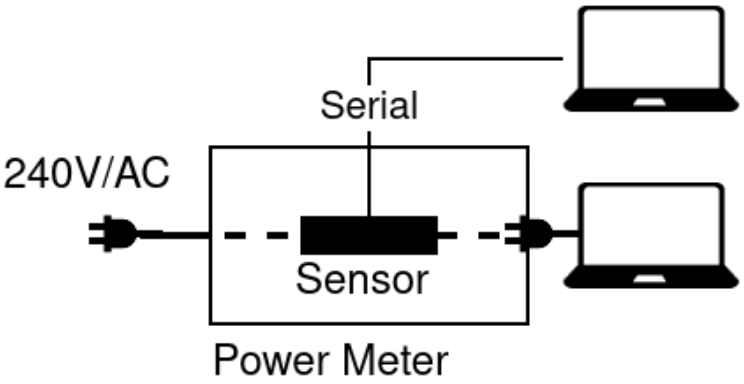


Figure 2.1: Overview of the setup using the hardware meter

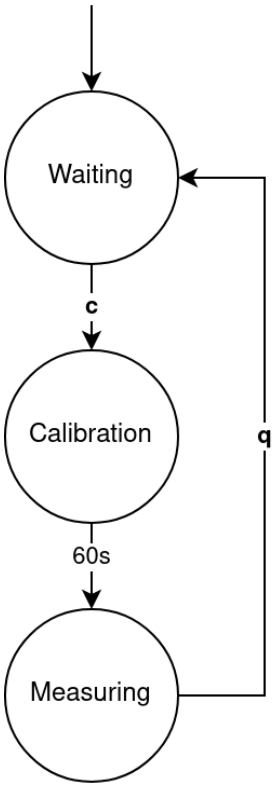


Figure 2.2: State machine of the hardware meter

```
struct HardwareMeterDatapoint {
      float instantaneousCurrent
      float instantaneousVoltage
      float activePower
      float apparentPower
      float fundReactivePower
}
```

Figure 2.3: C-struct data format for data sent by the hardware meter

2.3 Running Average Power Limit (RAPL)

Modern CPUs from Intel and AMD have interfaces that allow to manage the power consumption of the CPU. This also includes the possibility to read the current power consumption [9] of the package, the main memory or just one CPU chip. The interface was first introduced by Intel in 2011 [9]. AMD offers similar functionality via the same registers since 2017 [48]. Figure 2.4 shows a general system architecture and marks which system parts correspond to which RAPL domain. Every domain is measured and can be read individually. As the figure shows the *psys* domain includes everything that is part of the CPU including integrated graphics. The `package` domain includes the CPU, but not the eDRAM and PCH component. The main memory has its own domain named *dram*.

2.3.1 RAPL on Linux

On the Linux operating systems the RAPL domains are mapped to files under the path `/sys/class/powercap/intel-rapl`. For reading the power consumption of the system, the most important files (which are present for each mapped domain) are the following:

- `name`: The name of the domain, e.g. *package-0*, *dram*
- `max_energy_range_uj`: The maximum value that the sensor will report.
- `energy_uj`: The consumed power since the system started or the value exceeded the maximum value and was reset back to 0.

Figure 2.5 shows the pseudo-code for reading the current power consumption from the RAPL sensor every second. One important step is checking if the current value is actually larger then the previous value. If it is not, then the value went through an overflow and was reset back to zero, so the actual value is the max value plus the currently read value. The actually used energy is then just the difference between the current value and the last value.

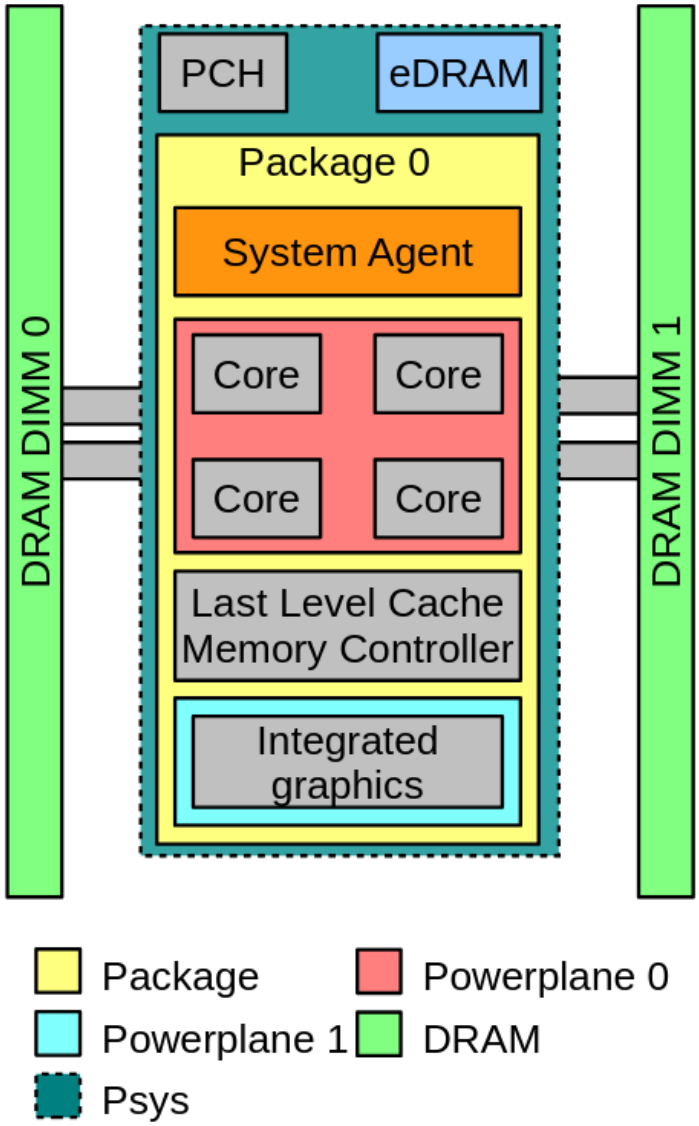


Figure 2.4: RAPL Architecture [27]

```
oldValue = read "./energy_uj"
while measuring {
      wait 1 second

      currentValue = read "./energy_uj"

      if currentValue < oldValue {
            currentValue += read "./max_energy_range_uj"
      }

      consumedPowerInLastSecond = currentValue - oldValue
      oldValue = currentValue
}
```

Figure 2.5: Pseudo-code for retrieving the current energy consumption from the RAPL sensor on Linux

Another possibility to read the RAPL sensor on Linux is to use the *perf* [42] tool. It is part of the kernel since 2009 [25]. The tool has a lot of performance related features, but the most interesting one for our purposes is reading the data from RAPL sensor. This can be achieved with the command `perf -a -e "<event of interest>" -I 1000`, which reports the amount of Joules that have been consumed each second for the given event. The list of available events can be retrieved with `perf list`, but the event *power/energy-pkg/* should be available on all systems which run on a CPU that has a RAPL sensor. It returns the value of the `pkg` domain.

2.4 Model-based Estimations

Since most consumers do not have external power meters that can measure the power consumption of a whole system (not just the CPU) and there is no standardized way to do this via CPU registers or similar mechanisms, many software based solutions fall back to a model-based approach. This uses some real data (e.g. the readings of the RAPL sensor, amount of RAM, current CPU usage,...) and estimate the power consumption of the components for which there is no direct way of measurement. In the case of the main memory this approach can be kept very simple. Getting the amount of RAM and multiplying it by a constant factor, would usually be a good enough approach. The main memory needs constant power to keep the data stored. The amount depends largely on the size of the memory. It can however also get more sophisticated by getting as much real data as possible and combining it with static information to estimate the energy consumption of the whole system. The data points could be something like the amount of RAM, the current amount of allocated memory and the amount of bytes that have been read from and written to the main memory. These values could then be used to estimate the power consumption of the main memory with models from prior research. This could increase the accuracy of the model, but is also more complicated. When

multiple parts of the system, i.e. CPU, main memory, hard drives, etc., are estimated like that and summed up, the power consumption of the whole system can be estimated. The real difficulty is however getting good data to base the models on. There are data sets available like the *SPECpower_ssj® 2008* [52], which is a list of server hardware and their power consumption. Based on that estimation models can be build for different hardware configurations. The *cloud-energy* project [18] is an example of a power consumption tool that does this.

2.5 System-wide vs Application-specific

The granularity of measurement has a significant impact on the difficulty of measuring energy consumption. The easiest case is measuring the whole system, since one can use a hardware power meter that sits between the power source and the system and reads the power directly. The difficulty starts when differentiating between multiple CPUs or parts of the system, like CPU, main memory, hard drives. Sophisticated setups allow to directly read these from compatible hardware, but these approaches need specialized equipment. On the software side it is more difficult. The RAPL sensor allows to get results for different parts of the system like CPU, DRAM, ..., but the used CPU has to support the different domains to be able to get measurements for them, which is not always the case. On consumer hardware there usually are just the two domains. *package* and the subdomain *core*. On professional-level hardware there usually is also the *dram* domain [21]. These are still not granular enough to read all the different components that make up a computer, like cards in the PCIe slots, hard discs, the cooling system, etc.

Additionally, not only the measurements for every system component are interesting, but breaking it down to the level of applications. A PC is usually not executing only one application at a time. This means that for each second for which the energy consumption is measured it should be divided up according to the amount of time each application has been running. Achieving this would mean that every process switch has to be monitored on all the cores of a multi-core CPU. This is of course not trivial and is also just an estimate of the real power consumption based on the amount of time the CPU was used by one application. Additionally this could skew the results since monitoring all this data can increase the energy consumption and it also needs CPU time itself. The Linux tool *PowerTop* [5] can divide the power consumption between processes, but only for devices that use a battery as their power source, so desktop computer and server hardware are not supported.

2.6 Related Work

The work that is most closely related to this thesis is the software that actually implements power consumption measurements. Table 2.1 shows a list of open source tools that are available at the time of writing. It presents the programming language, the possible ways

of interaction, the measuring granularity and the output format including the unit used for the values in the output.

Tool	Programming language	User interface	Granularity	Output info	Output format
RAPL [21]	Hardware	File	System	μJ	Number
perf [42]	C	CLI	System	J	Stdout
PowerJoular [35]	Ada	CLI	System Process	J	CSV Stdout
CodeCarbon [6]	Python	Python CLI	System Process Code	kWh $CO_2\ g$	Stdout CSV Rest API
Tracarbon [13]	Python	Python CLI	System Code	W $CO_2\ g$	JSON Stdout
Eco-CI [17]	Python	CI pipeline	Python build & tests	mJ	Pipeline output

Table 2.1: Comparison of energy consumption measurement tools

RAPL is a CPU sensor that allows to retrieve information about the energy consumption of system components like the CPU and main memory. The data can be read by either directly fetching the information from the CPU registers of by reading the mapped files in Linux (see section 2.3 for more details).

perf is a Linux tool introduced in 2009 [42]. It is a performance analysis tool on the command line that also allows to fetch the information from the RAPL sensor. The program allows to set which domain of the RAPL sensor to read (i.e. `package`, `psys`, `dram`, ...) and how often this should be done. The output is directly printed to the terminal. If the output needs to be used further, it needs to be captured, parsed and converted to an appropriate format.

PowerJoular is a command line tool written in Ada, which was introduced by Adel Noureddine in 2022 [35]. It can measure single processes and returns the data either as CSV file or directly printed to the terminal. As data source it uses the mapped files of the RAPL sensor. The consumed energy is measured and reported in Joules. PowerJoular can also measure the energy consumption of NVidia GPUs.

CodeCarbon is written in Python and can be used as a standalone tool or as a Python library. It offers the possibility to not only measure the whole system or one process, but particular lines of Python code. This can be useful when analyzing Python code in detail. Its output can be send to the terminal, a CSV file or to a Rest API. It outputs the energy consumption either as grams of CO2 equivalents or as Watts. The carbon

emissions are computed when the information of the country the computer is located in, is configured. The information about the electricity mix of the country is fetched from `https://co2signal.com`. It uses the RAPL sensor and the NVidia interface for graphics cards as data sources.

Tracarbon is also written in Python and can also be used as a standalone tool or Python library. In addition to printing data directly to the terminal, it can also be exported as JSON or sent to a Prometheus or Datadog backend. It also allows to compute the carbon emissions by using the *co2signal* API. Additionally the AWS API that returns the electricity mix of the used AWS node (if Tracarbon is executed on AWS) or the data from the European Environment Agency can be used for the carbon emission calculation. The data sources used for the energy consumption are the RAPL sensor and the NVidia interface for the energy consumption of GPUs.

Eco-CI is not a standalone tool that can be downloaded and used locally, but can be integrated into Github Actions pipelines to estimate the energy consumption of the execution of a pipeline. It captures the CPU utilization of the virtual machine the pipeline is running during the execution and uses the Cloud Energy project [18] to estimate the energy consumption of the machine. The underlying technique is a XGBoost model trained on the SPECPower dataset [52] which contains information about different hardware configurations of servers and their respective energy usage for a given percentage of CPU utilization. The Eco-CI tool cannot be used locally for measuring the power consumption, but by using the underlying Cloud Energy project, the energy usage of a system can be estimated by directly using the XGBoost model if the necessary information about the CPU are available.

What becomes clear from this is that the landscape of available tools is quite heterogeneous. The output units are not the same. Some tools offer a command line interface and some tools offer a library that can be used from within another program instead or additionally. If researchers or engineers wanted to compare these tools or measure the same piece of code or software with multiple ones, then significant effort is needed to get the data from these tools and transform it into a comparable format.

CHAPTER 3

Method

This chapter first presents the goals that have been defined for the framework, followed by an overview of the system architecture that has been implemented to achieve these goals. Last, we discuss the three most interesting parts of the framework in detail. The interaction between the test machine and the tester in subsection 3.3.1, the wrapper structure in subsection 3.3.2 and the testing process in subsection 3.3.3. We will also discuss how these fulfill the initially set goals.

3.1 Goals

The following goals have been defined for the CoMECS framework.

Openness The framework should be able to test different hardware. It should not be a solution built for a single or a small number of test machines that have been setup in a lengthy process just for this thesis. Additionally, the computer that runs the framework should not need a complicated setup process. Defining additional tests or adding more power meters should be possible in an intuitive way that allows researchers to test their own software in the future.

Reproducibility The tests that are defined within the framework should be able to be executed many times without needing manual setup steps between the runs, even when other machines are tested than the one that was used when initially creating the wrappers and tests.

Standardized Output All the output of the framework should be standardized and unified in a way that can be used for further analysis and comparison without additional manual effort. The format should be human- and machine-readable.

If a framework succeeds in fulfilling these goals, it allows to easily run tests with a wide array of different power meters using hardware of their choice, which allows researchers to compare not only software like web servers, but also power meters themselves against each other in a well-defined way. A properly designed and open framework would not only allow to test different predefined software, but allows more complex tasks without a lot of additional effort. These can be something like executing the same test against different versions or commits of the same program.

CoMECS is the result of our effort to realize this vision. It tries to give researchers a powerful tool to measure the energy usage of software.

3.2 Overview

Figure 3.1 shows the overview of the process that is executed for each experiment that is done with the CoMECS framework.

The process is split into three main sections. *Preparation*, *Testing* and *Postprocessing*. The *Preperation* phase prepares all the necessary files for the second phase and uploads them to the test machine. The test is then executed there in the *Testing* phase and the results sent back and processed into the final result in the *Postprocessing* phase.

Preparation The input is comprised of a configuration file and command-line arguments. Both can configure the same values, except for the list of tests to execute and meters to use. These have to be given via arguments. The command-line configuration also takes precedence over the configuration file, which allows to prepare one configuration file that functions as base and overwriting the parameters that change in-between the experiments via the command-line. From the given parameters in the configuration, the parameter combinations are computed by building their cross product. This is done for every given test/meter combination. Therefore CoMECS only creates parameter configurations, that are actually needed for a test run. This lets the framework only execute tests that are unique in their configuration and therefore avoids unnecessary computing resources by just building all possible combinations. The concrete wrappers are then compiled from their templates, which is described in section 3.3.2. The compiled wrappers are then uploaded to the test machine. The test machine is now prepared for the actual measurement process, which is described in section 3.3.3.

Testing During this phase the previously uploaded wrappers are activated, which install all the necessary dependencies for running a test. The execution then produces the raw outputs of the measurement with the chosen power meter as artifacts. These have not yet been converted into the standardized format. They are then transferred back to the executor machine for the next step.

Postprocessing The raw output files are converted into the standardized format. This is done by the converter script that is defined for each wrapped meter.

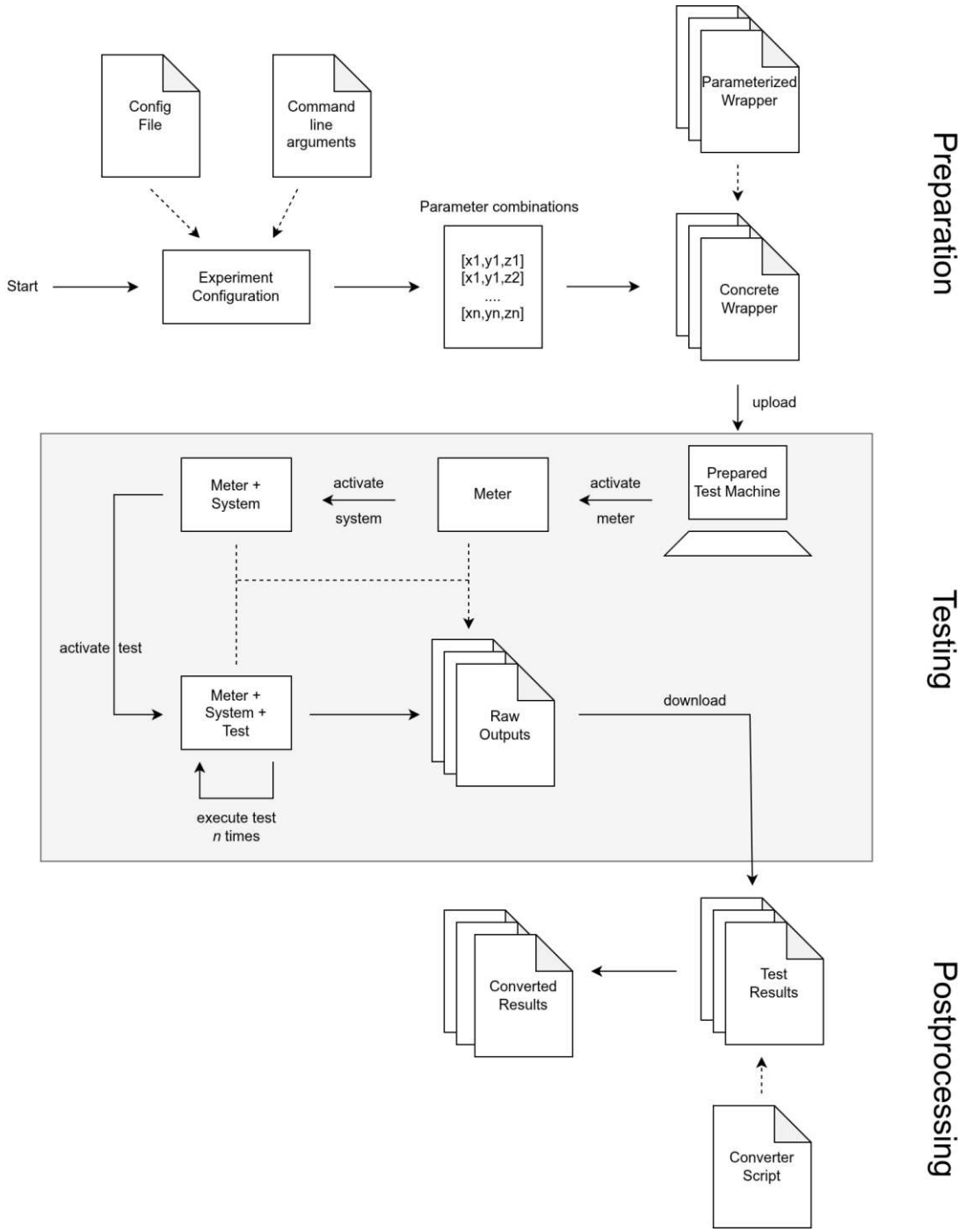


Figure 3.1: Experiment process overview

This whole process is done for every combination of test and meter defined by the command-line arguments.

Figure 3.2 shows how the involved machines have been setup and which techniques are used for the communication between the different components.

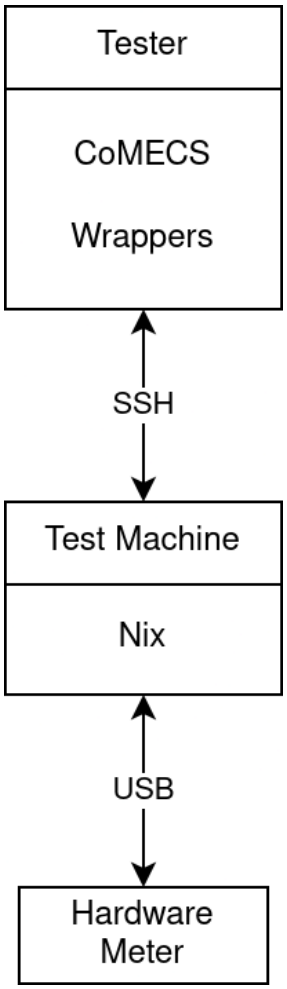


Figure 3.2: Test Setup

The basic setup for CoMECS consists of a tester or test executor and a testee or tested machine. The tester needs to have the framework installed, including the wrappers and be able to connect to the tested machine via SSH. All of the testing and measuring is done on the tested machine and then downloaded from there via `scp`. The measurement captures the output from the tools in their respective format, but the transformation is then done on the tester. So there is minimal overhead for the tested computer.

Some meters might require more resources than others while measuring, which would skew the results just by collecting data. Thus, we offload them to other machines that

then run the measuring process completely separated from the tested machine. Figure 3.3 shows this kind of setup. In this architecture the hardware meter has two ports. One for

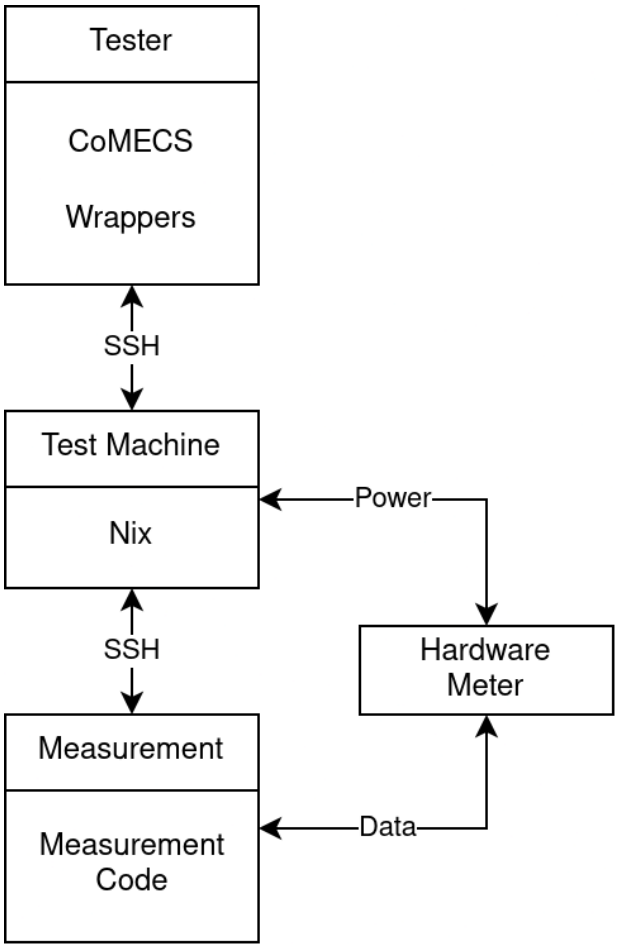


Figure 3.3: Test setup with an additional measurement machine

power and one that allows to read the data. The power port is still used by the *Test Machine*, but the data port now connects to an additional *Measurement Machine*. The tested computer sends commands via SSH to control the starting and stopping of the measurement on the extra computer and to download the data, instead of executing the meter locally. This allows to gather data that is not skewed by the additional effort of running the meter. This is however not possible with all meters. We therefore did not design CoMECS with this architecture as default, since all power meters except for the hardware meter have to be executed on the same machine that is also executing the tests. However the implementation of this more sophisticated setup did not take any changes in the implementation of framework itself.

3.3 Details

In the following sections the most interesting components of the framework are presented. Section 3.3.1 describes how the interaction between the executor and the test machine is accomplished. It describes how an SSH connection is used like a raw socket to allow to start shells within the initial one and what tricks are needed to still have the synchronous communication that SSH is used for. In section 3.3.2 the structure of the wrappers are described and how they can be parameterized. The test process including the three measurement steps and how the actual results are then computed from the raw output of the meters is described in section 3.3.3.

3.3.1 Interaction

The tested machine is controlled exclusively via SSH. This protocol is extremely popular and nearly every operating system has a readily available implementation of an SSH client and server that can be installed easily. It allows to start a remote shell on a machine through an encrypted connection and execute commands as if it they were executed within a local shell.

The only problem with SSH is that it is designed as a request/response style of communication or to start a shell remotely and use it interactively. Both of these styles do not really allow to automate multiple steps that need to be executed in varying layers of shells. Since executing the command to launch a shell remotely, blocks until the shell is closed again. This is because SSH client implementations usually execute a command, wait for it to finish and return the result. This behavior therefore does not allow to execute commands programmatically within a second shell started in the SSH session, because a shell does not finish and return a result until it is closed. Therefore a wrapper was written that starts a connection via SSH, but then uses it as if it was a raw socket. It takes the input and sends it to the remote shell and outputs the shells result. Unfortunately all the inputs to standard in are reflected back on standard out and both of these streams are written concurrently without built-in synchronization. We solved this by using markers to signify the start, end and return code of a command. Our SSH wrapper also encodes all commands with base64, which are decoded ad-hoc on the remote system to avoid problems with quotations or special characters. Every command is wrapped in the structure seen in figure 3.4. This allows to synchronize input and output and to get the full result of the executed command including the return code. On the left side of the figure there is the snippet of shell script that is executed for a single command, that is executed synchronously on the remote machine and for which the result is then returned. Instead of sending every line on its own, which would result in executing them one after another, the lines are separated with a semicolon and send as one single command. This results in an uninterrupted output stream that only contains the results and none of the inputs reflected back in-between. The bottom sub-figure shows the result that can be observed in the output stream, when executing the shell code in the top sub-figure as described.

```
echo <start-marker>
<command>
RETURN_CODE=$(echo $?)
echo <return-code-marker>
echo $RETURN_CODE
echo <end-marker>
```

(a) Executed commands

```
<start-marker>
<command-output>
<return-code-marker>
<return-code>
<end-marker>
```

(b) Produced output

Figure 3.4: Wrapper for a synchronous shell command

This approach however only works for commands that run until stopped. When using just that command wrapper the functionality is the same as with all the other SSH client implementations. For being able to start a shell within a shell (or environment) another wrapper is needed. The wrapper structure for activating an environment (or a shell within the shell) is shown in figure 3.5. In addition to a start and end marker this also includes a fail marker, which signifies an error during activation. It also includes a setup code, which deactivates unnecessary decorations which would make the shell more graphically appeasing for a user, but in the case of automation just results in more output bytes which will be ignored anyway and need to be explicitly removed when parsing. The used setup code to remove these decorations is `export TERM=dumb;export PS1="`. The `TERM` environment variable is set to `dumb`, which signals the shell that the terminal does not support any decorations or advanced features, besides plain ASCII text. The `PS1` environment variable is set to an empty string. This causes the shell to not print a prompt like `[user@hostname ] $`, which would show up in the command output and needs to be removed.

Contrary to the command wrapper the snippet to start a shell is sent as three separate lines instead of one single line. The first line starts the shell or and or prints the fail-marker. The second line executes the setup command to remove all decorations and the last line prints the end marker. The SSH wrapper can therefore wait until it sees the fail-marker or end-marker and knows that the command to start the shell failed or that is successfully opened now. In case of success all the following commands are then executed

```
echo <start-marker>; <command> || echo <fail-marker>
<setup-command>
echo <end-marker>
```

(a) Executed commands

```
<start-marker>
<activation output>
<end-marker>
```

(b) Produced output on success

```
<start-marker>
<error output>
<fail-marker>
```

(c) Produced output on error

Figure 3.5: Wrapper for activating an environment

in the newly opened shell. An environment started by this can be easily closed again by sending the `exit` command (wrapped in the command wrapper), which will close the currently running shell. This approach allows to freely open new shells within shells and also close them again in a synchronized fashion, that allows to wait for commands to finish and communicates back the success or failure state of the last executed command, no matter if it was a terminating command or a command that opens a shell.

3.3.2 Wrapper Structure

The CoMECS framework allows adding tests, systems and meters by wrapping them in a particular structure. This wrapper allows the CoMECS to interact with all tests, systems and meters in the same manner, which makes the automation of test executions very easy. All three types have a config file (`config.toml`) and an environment file (`shell.nix`) in common. Meters also have a wrapper script (or command) and a converter script. The wrapper structures for tests, systems and meters are shown in figure 3.6.

The following two files need to be defined for every wrapper, no matter its type.

- **config.toml** contains the name and the description. Meters also can have a flag if there is a converter script, a list of template variables and a calibration time,

```
<test>
├── config.toml
└── shell.nix

<system>
├── config.toml
└── shell.nix

<meter>
├── config.toml
├── shell.nix
├── wrapper
└── converter
```

Figure 3.6: Wrapper Structures

that can be used to set a wait period before the meter is ready. Tests need the name of the system they rely on, additionally to the base configuration. Figures 3.7 and 3.8 show the files for *Tracarbon* power meter and for the test which executes the *pgbench* tool for the *pgbench-baremetal* system, as examples of `config.toml` files.

- **shell.nix** contains the shell environment with all the dependencies the wrapper needs to run. For systems this file also includes a `shellHook` script, which is run when activating the shell. It starts the system and stops it when a SIGINT signal is received. The signal handler also needs to clean up created files or resources, so that the system is in the same state before and after the system was executed. Figure 3.9 shows the file for the *nginx* wrapper, which includes a custom *nginx* package and a `shellHook` which defines a cleanup function to properly stop the started processes and cleanup all remaining temporary files.

```
[general]
name = "Tracarbon"
description = "Tracarbon tracks your device's energy
    consumption and calculates your carbon emissions
    using your location"
url = "https://github.com/fvaleye/tracarbon"
converter = true
calibration-time = 4
```

Figure 3.7: `config.toml` file for the *Tracarbon* meter

```
[general]
name = "pgbench-baremetal"
description = "This runs pgbench on a Postgres instance
    running baremetal."
command = "pgbench -T 30 -U postgres -h localhost
   postgres"
system = "postgres-baremetal"
```

Figure 3.8: `config.toml` file for the test, which executes *pgbench*

```
{ pkgs ? import <nixpkgs> {} } :
let
custom-nginx = pkgs.callPackage ./nix/custom-nginx.nix
   { };
in
pkgs.mkShell {
      nativeBuildInputs = [
      custom-nginx
      ];

      shellHook = ''
      function cleanup()
      {
            kill $(pidof nginx)
            rm -R /tmp/nginx*
      }
      trap cleanup EXIT
      nginx
      '';
}
```

Figure 3.9: `shell.nix` file for a custom *nginx* package

For wrapped meters additionally these two files need to be defined.

- **wrapper** is an executable that starts the meter. It has to start in the foreground and it has to react to a `SIGINT` signal by stopping and writing out its result. The path to the result file is given as first command line argument to the executable.
- **converter** is an executable that gets two command line arguments. The first one is the path to the input file and the second one the path to the output file. The program is supposed to convert the input file (which is created by the wrapper script) into the correct output format defined by CoMECS.

Additionally to these files there can also be extra files. The whole wrapper directory is uploaded to the test machine. Thus, the *shell.nix* or the *wrapper* executable can use files

which are located within the wrapper directory itself. This allows the most freedom for structuring the files or if there are additional files needed, like a custom nix package for a program.

Examples of all three of the wrappers are shown in appendix 6.1.

Templates and Parameterization

CoMECS not only allows to create static wrappers, but also define them as templates with placeholders. Before uploading the wrapper directory to the test machine, all files with names ending in `.template` are considered to be templates compiled into concrete files. They can contain placeholders (*template variables*) that will be filled in by the framework with the parameter values defined in the run configuration. As template engine the built-in Python *Template Strings* [45] are used. They allow to define placeholders that can be filled by static values. Expressions or more advanced features like conditionals or loops are not implemented. Before uploading a wrapper to the test machine, the framework copies the full wrapper directory and iterates through the whole file tree. When a template file is found, it is copied, the `.template` file ending removed and all *template variables* are replaced with their real values. The compiled directory is then uploaded to the test machine. The real values are found by first checking which variables are defined in the configuration files of the meter, system and test and then creating their cross product. Then every test is executed with every combination of these parameters. This templating feature facilitates dynamic wrappers that allow to implement use cases like different versions of the same program or parameterized tests with a different amount of iterations executed.

Figure 3.10 shows how the parameter configurations are built from the defined parameter values, the tests and meters, and the parameters they are defined to use. On the left side of the graphic, there is the list of defined parameters (a, b, c, x, y, z) and their values $(a1, a2, ...)$. In CoMECS this is done in the configuration file or the command line arguments. On the top there are the defined tests (A and B) and meters (X, Y). The parameters they use are shown as diamond shaped tags. From the list of tests and meters the cross-product is built into the combinations $AX, BX, AY \text{and} BY$. The parameters these combinations use are just the concatenation of the parameter lists of the test and meter the combination was built from. The list of parameters of all possible parameter configurations is then used to build the parameter combinations. In the the given example this would result in a total of 72 tests instead of 216, which the cross-product of all the available parameters would result in. From these 216 combinations 145 would therefore be unnecessary tests.

All the configurations of the form

$$a1, bn, cm, x1, yk, z1$$

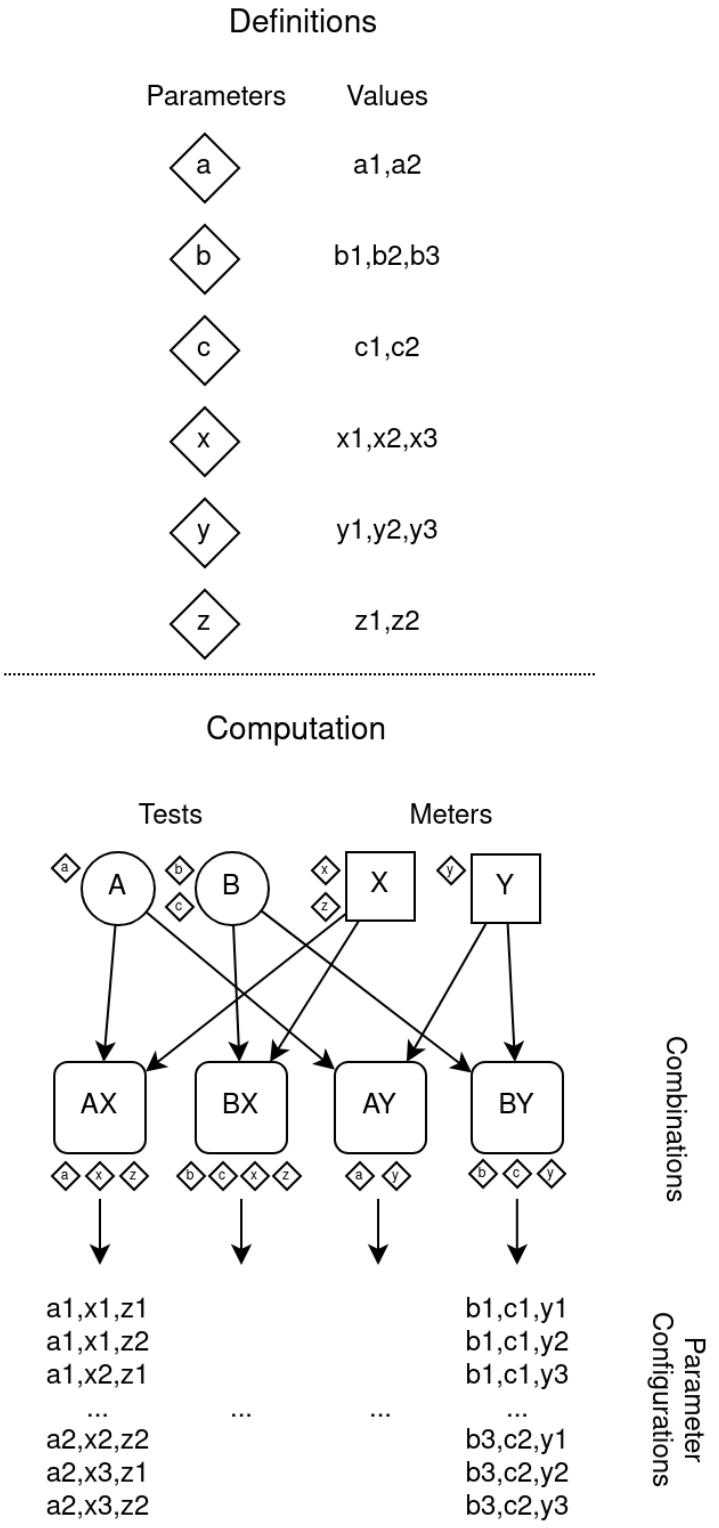


Figure 3.10: Computation of parameter configurations

where

$$bn \in \{b1, b2, b3\}$$
$$cm \in \{c1, c2\}$$
$$yk \in \{y1, y2, y3\}$$

are the same test, when executed on the AX combination, since bn, cm and yk are unused.

They are not adding any new information of to the experiment, since the effective parameters (the ones that are actually used) would be used for multiple executions that will yield the same result multiple times. By not including these parameter configurations we not only save on time, but also on computing resources.

The example simplifies a little bit, since it does not include systems, even though they can also be parameterized. They are however not relevant for the building of the test/meter combinations, since the tests define which system they rely on. The parameters defined by a system wrapper are therefore just added to the ones the test defines. This list of combined parameters are then considered to be test parameters.

This templating and parameterization system reduces the need for duplication since it allows to defined one wrapper that fits multiple experiments except for one detail, which can then later be filled in during the execution instead of having multiple duplicated wrappers that have only this small detail changed.

Nix

One of the core technologies used for the wrappers and which is an essential part of the test process, is Nix [33]. The particular part of Nix, that is used for the purpose of this framework, is the possibility to easily create and destroy environments with a defined set of packages (i.e. wrapper dependencies). This technique is called *nix shell* [34]. By defining the wanted dependencies in a `shell.nix` file in a declarative manner, a user can call `nix-shell shell.nix` (or just `nix-shell` if the file is called `shell.nix`) to start a new shell that has the dependencies installed. When exiting the shell the dependencies are removed again. This makes it really easy to install and remove software when needed. Another feature that makes it so useful for our purposes is that these environments can be started within each other. During the test process first the environment for the meter is activated, after that the one for the system. Within the new shell that contains the dependencies for the system, the ones from the meter environment are also available. The ones from the test wrapper are however not installed. This means that by layering these shells, the framework can selectively activate and deactivate the currently needed packages. At every point in time only the necessary packages are installed and nothing else. Additionally there can be a `shellHook` section defined, which is a shell script executed when the environment is activated. For system wrappers this is used to start the currently tested system, e.g. a PostgreSQL database or nginx. By using the bash command `trap` the executed bash script knows when the shell

environment is deactivated and can stop the system and clean up resources not handled by nix itself, e.g. created database files or temporary files. Since the `shell.nix` files are declarative and use a central repository hosted by nix, they are reproducible on different systems that have a functioning Nix installation, without having to change anything in the `shell.nix` file itself. This allows to effectively share these files with the research community.

3.3.3 Test Process

The CoMECS framework is used through a command line interface, which allows to define and execute *experiments*. These can include multiple *tests* that are measured with multiple power *meters*. Every test/meter combination will be executed sequentially on the test machine. Additionally if there are variables defined, then every possible variable combination will be computed as described before. Figure 3.11 shows the process that happens when one variable configuration is executed for one meter/test combination and what environments are active for each phase of a test. First the meter environment is activated and a baseline measurement is taken. Then the system environment is activated additionally and a system measurement is taken. Lastly the test environment is activated and a test measurement is taken for each execution of the test. A measurement means activating the power meter waiting for the test to finish or 30 seconds to pass and stop the meter again.

The results from this process are downloaded and every file containing the output from the wrapper script are converted to the correct format using the corresponding converter script. The output format is defined to be a CSV file with the following columns:

- **Tick** The second after starting the measurement
- **Joules** The amount of joules that have been consumed as measured by the meter for this tick
- **Cumulative Joules** The amount of joules that have been consumed since starting the measurement

After converting the results to their correct format an additional step is necessary to get the actual energy consumption of the test, since the measurements from the test still also contain the energy consumption from the underlying system. First the median baseline consumption is computed and then for each run this value is subtracted from each tick of the test measurement. The result is written into its own CSV file. All the CSV files are put into a directory for that particular test so that further analysis on the raw results can be done by the researchers if needed.

The resulting file structure is shown in figure 3.12. The root directory is either the timestamp of when the test executor started or a configured experiment name. For every test there is a subdirectory, which contains a directory for every meter that is

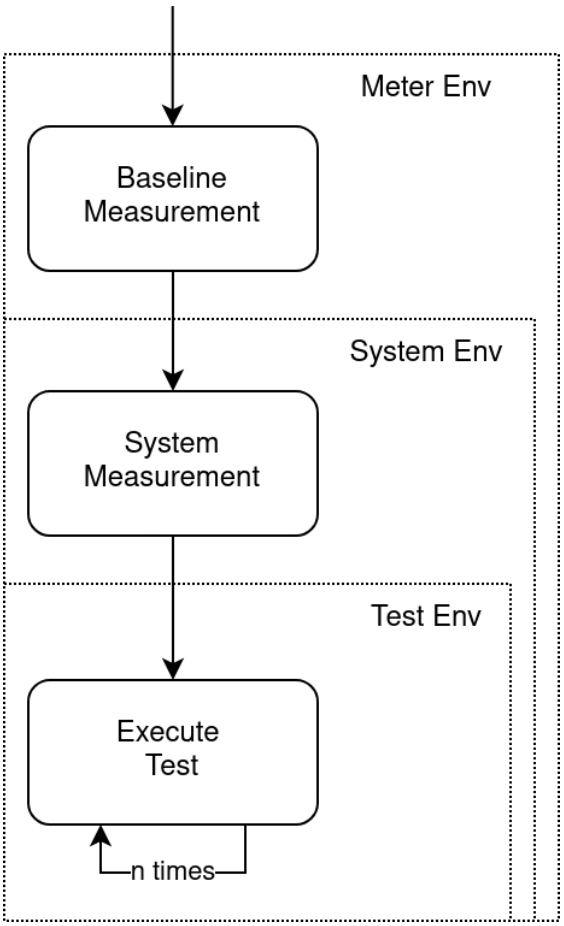


Figure 3.11: Test Process

```
<experiment>
└─ <test>
   └─ <meter>
      └─ <parameter-config>
         ├─ baseline.csv
         └─ system.csv
            ├─ runs
            │  └─ 0.csv, ..., n.csv
            └─ results
               └─ 0.csv, ..., n.csv
```

Figure 3.12: Test Result File Structure

executed for this test. If there are parameters used for wrapper templates, then there will also be one directory for each executed variable configuration, which has the format: `var1=value1,var2=value2`. Every executed meter-test-parameter combination contains a `baseline.csv` file, that contains the results of the baseline measurement, a `system.csv` file, which contains the result from the system measurement. There is also a `runs` directory, that contains the CSV files for the n runs and the `results` directory, which contains the CSV files for the n results that where computed from the data of the runs and the baseline as described before.

The directory structure is always the same, so extra post processing steps like generating diagrams is also reproducible and can be automated in a reliable way, even if it is not handled by CoMECS itself.

3.3.4 System requirements

After presenting the most important features, the actual system requirements for running CoMECS are presented in this subsection.

Test Machine The tested machine needs to have an SSH server running and nix installed. Additionally there need to be two users that are available to the runner. The root user for preparing the wrappers on the host and an unprivileged user that is executing the tests. Some setup steps are done by CoMECS itself, like generating a directory structure that is used for uploading the wrappers, but these steps are universal for all meters. However some wrappers might need a specific preparation, which can include allowing nix to use packages labeled insecure or allowing non-root access to the RAPL sensor. These have to be prepared manually beforehand or added to the wrapper itself, if it is possible to do them with the unprivileged test user.

Framework Executor The machine that executes the framework needs be able to run CoMECS, which is written in Python. The project needs to be setup, which can be done by following the steps in the `README.md` file. Additionally a SSH client and connection to the *Test Machine* is required. It of course also needs to have the prepared wrappers for the meters, tests and systems that are used for the experiments.

These system requirements are fairly minimal and are easily prepared on different hardware. Also once a server is prepared different CoMECS instances can use it to run experiments as long as they have the required SSH access to the server.

3.3.5 Configuration

The execution configuration for CoMECS has to sources. The static config file `config.toml` in TOML format and the command-line arguments. Every setting can be defined in either of them. Only the list of tests to execute and meters to use can not be defined in the configuration file. When CoMECS starts it first reads the configuration file and parses it using the `tomli` [19] library. The resulting Python dictionary is then checked against a

JSON schema [26]. If it does not pass the check, the frameworks stops. After that the command line arguments are parsed by the `argparse` [47] library. Every value that is given as command line argument is then applied to the previously parsed configuration and overwrites already defined settings. This allows to define a base configuration and overwrite it with specific settings for a certain experiment.

The `config.toml` files that every wrapper needs have, are also in TOML format and go through the same parsing steps. First they are parsed and then checked against a JSON schema. If they do not parse the check, then they are skipped and a warning is written to standard out. This shows that there are problems with the wrapper, so the user can fix them, but long running experiments (many tests, meters or repetitions) are not interrupted by one wrong wrapper.

The JSON schemas used are listed in 6.3.

3.4 Discussion

3.4.1 Challenges

Besides the implementation specific challenges already discussed, there was another challenge that came up during the testing phase was that we noticed a strange behavior on the server hardware when using the hardware meter. The results from the hardware meter compared to the RAPL sensor through `perf` suggested that the RAPL sensor overestimates the energy consumption. This however should be nearly impossible, since the RAPL sensor is embedded in the CPU and even the `psys`-domain should still underestimate the power consumption compared to the one that is measured by an external power meter. Additionally the problem did not come up during tests on the desktop machine.

Our investigation found that the hardware CPU had a baseline power consumption of around 30W and when running a specific benchmark consumed up to 70W. So the difference of 40W is what CoMECS calculates as the power the benchmark is using. We know these numbers, because the server contains a power measurement sensor and the management dashboard regularly presents the current numbers of that sensor. The test using the RAPL sensor returns numbers similar to those of the dashboard, so we knew that they have to be correct. The hardware meter however returned 50W for the baseline measurement, but 70W during the test run. So the difference came out to just 20W, instead of the expected 40W.

Our first suspicion was that the program that connects to the power meter and reads the data is to blame for this increase. Since the hardware device sends 80kb per second (see figure 3.13) and the reading program is implemented in Python we suspected that the inefficient choice of programming language caused increased CPU and main memory usage, which in turn increased the CPU temperature and caused either additional cooling to start or the CPU to increase the clock speed. This effect however seems to only occur when the CPU is in not under load, but disappears if the CPU is already under load.

$$
\begin{aligned}
5 \text{ floats} * 4000Hz/s = \\
5 * 4 \text{ bytes} * 4000Hz/s = \\
20 \text{ bytes} * 4000Hz/s = \\
80000 \text{ bytes}/s = \\
80 \text{ kilobytes}/s
\end{aligned}
$$

Figure 3.13: Calculation of the amount of data the hardware meter sends

The solution to this problem was to reimplement the program in a more efficient language. We used Go [16] for this, since it it a high level language that compiles to a binary and has a small memory footprint and good performance, especially compared to the interpreted Python language. This however resulted in the same effect. Even when optimizing memory allocations by preallocating an array that can store all the data from a second and optimizing memory reads to be consecutive by not unpacking the data as it comes in, but after a second worth of bytes are read and then unpacked all at once.

Our current hypothesis is that the amount of data that is sent over the serial port and the resulting memory reads and writes increase the CPU load enough to cause a higher clock frequency or temperature and therefore additional cooling, which in turn causes the energy consumption to increase. For this reason we used the setup shown in figure 3.3.

3.4.2 Ansible

Another approach to automation which has a lot of overlap with what CoMECS is trying to achieve is Ansible [4]. It allows to define tasks that are executed on a remote machine with not much more than a SSH connection. These tasks include installing and uninstalling dependencies, executing scripts and returning their result, and many more. The difference between Ansible and CoMECS lies in the *not much more* part. Ansible's approach is to build a Python script, which is then sent to the remote machine and executed there. This means there is the overhead of running Python on the remote machine, which controls all the other executions. CoMECS executes everything directly in the shell. Even the created nix-shells are nothing more than wrappers around downloading dependencies, which are then "installed" by setting environment variables in a way that the dependencies can be found by programs as if they were installed in the normal way. This means there is nothing running in-between the shell and the executed test. Since our goal is to measure the energy consumption, having anything running which is not explicitly wanted would skew the results. There is also no easy way to start environments and stop environments. So either Ansible playbooks have to be created dynamically based on the experiment or shell script, which does the same thing that the SSH wrapper of CoMECS does (i.e. starting and stopping environments programmatically including returning errors and stopping the execution). This approach however then

ruins all the advantages that Ansible offers and would reduce the work that it does to uploading and executing a script, which would be easier to do with `scp` and `ssh` directly. Ansible is also not fully distribution agnostic. It is possible to write playbooks that are handling dependencies in a distribution agnostic way, but this needs a lot effort again. One could argue that *nix* for CoMECS handles the same use case as *apt* for Debian-based distributions. The difference is that *nix* can installed on most available Linux distributions in addition to existing package managers, which is not the case with *apt*.

3.5 Conclusion

In this chapter we presented and discussed all the decisions made during the implementation of CoMECS. The described steps and considerations have been guided by the goals set at the beginning of this chapter. The next paragraphs evaluate each goal and discuss why we consider them to be achieved by our framework.

Openness CoMECS uses SSH and Nix, which are not restricted to a single operating system. Wrappers based on Nix can package any software available for Linux and it has an extensive library with over 120 thousand freely available packages. The parameterization of wrappers adds a powerful tool to write reusable and adaptable tests. Relying on well-tested, widely available and popular technologies allows to use any hardware that is capable of running Linux as test machine and every system that can run Python as test executor. Additionally the parameterization allows to create wrapper templates for which details, e.g. a specific version or an API token needed for access to a private repository, can be configured during execution.

Reproducibility Relying on declarative shell environments for dependency management, makes it possible to reproduce the tests on any machine that has nix installed. This allows to write distribution-agnostic wrappers that can be executed quickly on a wide range of already prepared machines without the need for a specific setup process for each computer. It also allows to share wrappers between users of the CoMECS framework.

Standardized Output The decision to use converter scripts that are forced to output a particular format fulfills the goal of having a single format for the output of the wrapped power meters. Before researchers had to either convert the outputs from wrappers by hand or rely on the power meters themselves to give them usable outputs. Often the decision was to write something similar to the converter script anyway, in which case CoMECS does not cause any additional overhead. The fixed format however supports the writing of these scripts, since there is no need to first decide on a specific format. It also makes post processing steps easier, since the format is already known beforehand.

CHAPTER 4

Evaluation & Results

In the following section we will present how the CoMECS framework was evaluated. First, we present which power meters have been wrapped and how much effort needed to be put in for the different programs. After this we discuss how they have been compared in three benchmarks that focus either on the CPU, memory or sequential disc writes. The results show which software meters are comparable to the hardware meters for what types of workloads. Second, we present the results of executing the same benchmark against a containerized version of the PostgreSQL database and a non-virtualized one and compare these results with the findings of previous research. Last, we show how we used the templating capabilities of CoMECS to define a series of tests with just a single wrapper, by comparing the energy usage of 16 different releases of the nginx webserver.

All the tests have been executed ten times on server and consumer/desktop hardware. The specifications of the two machines are listed in table 4.1.

	CPU	RAM	Harddrive
Server	2x Intel(R) Xeon(R) Gold 6126	4x DDR4 64GB	2x SATA SSD 250 GB
Desktop	Intel(R) Core(TM) i7-8700 CPU	1x DDR4 16GB	NVMe SSD 500GB

Figure 4.1: Hardware Specifications

4.1 Power Meters

The power meters that we chose to wrap for CoMECS are *perf* [42], *PowerJoular* [35], *CodeCarbon* [6], *Tracarbon* [13], *Cloud Energy* [18] and a hardware meter from a previous master thesis [8]. This hardware-based power meter is used as a baseline to compare the software-based tools against. Since these power meters are vastly different in how they are provided and used, they had quite different requirements when wrapped for our framework.

perf The *perf* tool is part of Linux since 2009 [25] and is therefore also readily available as a package for Nix. The wrapper can be reduced to a single line. The command we used is `perf stat -a -e "power/energy-pkg/" -I 1000 2> $1`. This reads the values from the `pkg` domain from the RAPL sensor every second and writes the result to the file given as the first command line argument.

PowerJoular This program was not available as package and needed a custom Nix package (called *derivation*), which described the installation that would need to be manually otherwise or automated using a Makefile or similar mechanics. The *derivation* defines `gprbuild` and `gnat` as build dependencies, where to get the source code from and the build and install steps as described in the `README` file of *PowerJoular*. The actual wrapper script is again only one line: `powerjoular -f $1`. The version that we wrapped for this thesis was `0.7.2`. Since writing the wrapper, *PowerJoular* has been added to the library of nix packages, starting with version *0.7.3* [15]. So the effort for the wrapper would be reduced to writing a shell script with the previously mentioned line.

CodeCarbon This software is not available as Nix package, but since it is available via *pip* [46] and uses standard build tools, the derivation for it is fairly simple. It only defines the Python dependencies and which pip package to fetch. The wrapper calls a Python script that uses *CodeCarbon* as library.

Tracarbon This Python tool is available as pip package, but because of assumptions that the nix Python helper functions make, the built-in derivation wrapper for Python packages, that was used for *CodeCarbon* could not be used here. Instead a custom derivation needed to be written that directly downloads the source code from the code repository instead and builds it from there. Additionally two dependencies needed to be packaged into custom derivations as well, since they were not available in the Nix library. They have however been added to the library, since we created the wrapper for the *Tracarbon*. The actual wrapper script is a Python script that uses *Tracarbon* as library.

ols & xgb These are two flavors of the models that the *Cloud Energy* project is based on. They are provided as Python scripts in a source code repository, that are supposed to be run within the downloaded repository directory and are not distributed through *pip*. They needed a custom nix derivation, which patches a hard-coded path in the script and a custom `setup.py` file, so that they can be installed as a Python program and be executed standalone. Additionally, they need to get the CPU utilization provided as input. The source code repository contains C code for such a program. This program needed to be packaged additionally to the meter itself. The wrapper scripts for both of these are just a one-line shell script, but they both need some information about the hardware of the system, especially the CPU, to function properly. These are provided via command line arguments in the script directly.

Hardware Meter This tool consists of the meter itself, which has a USB port and electric outlet. These have to be prepared and connected to the correct computers beforehand. The interaction with the meter is done via serial port through the USB connection, using a Go program that runs the correct steps to interact with the hardware tool. Because of problems that have been discussed in chapter 3, this meter is different from the rest, in the sense that the computer that receives the data from the measurement is not the same as the PC that generates the data. The wrapper for this meter is therefore a shell script, which is executed on the test machine, that runs commands on the measurement machine via SSH. The measurement machine has to be prepared beforehand with all the necessary dependencies to be able to start and stop a measurement. These are shell scripts that control an instance of the Go program that interacts with the meter via serial port.

The converter script for all the meters is fairly similar. It is a Python script that scans through the output lines and uses regular expressions to parse the data from the output lines. Some meters need conversion from *Watts* to *Joules* or similar, but all of this is very simple for ticks which are one second long. The only exception is the hardware meter which outputs not only the used energy but also a lot more properties of the electrical energy flowing through the meter. This is done with a resolution of 4000Hz, which needs to be compacted down to one value per second.

The wrapping of the meters is mostly the same effort as would be needed when installing and using them manually, if the knowledge of Nix is already there. However the difference is that the declarative packages can be reused and shared easily and capture all the necessary dependencies explicitly. They also capture and store the knowledge that would easily be lost, when doing this setup by hand. In that case one would most probably install the dependencies as the need arises and forget which exact dependencies where needed over time, if no proper care is taken to note these steps. Additionally the dependencies might not be directly transferable to different distributions, because they are named different in the repositories of the respective package managers. Additionally the written shell script might not be easily transferred to other machines, when assumptions of the underlying system are hard-coded. It is possible to take steps to not make these mistakes, but it might still not be compatible or comparable with other setups that achieve the same result. Therefore the structure that CoMECS enforces allows the most freedom, but still gives enough structure to make everything interchangeable.

4.1.1 Comparison

One of the questions that CoMECS tries to answer is how different meters compare to each other with various workloads. To answer this, we used the *sysbench* [2] benchmarking software. It provides tests that focus on different parts of a system. We decided to use tests for the CPU, main memory and disk input-output as these are three of the most important aspects of computing. The used *sysbench* invocations are:

- *CPU*:

```
sysbench cpu -threads=12 run
```

- *Memory*:
  ```
  sysbench memory -threads=12 run
  ```

- *Sequential writes*:
  ```
  sysbench fileio -threads=12 -file-test-mode=seqwr run;
  rm test_file.*
  ```

The figures 4.2,4.3,4.4 show the measurements in the server environment. They show that *perf*, *CodeCarbon* and *Tracarbon* follow the results of the hardware meter closely with a bias of around -200 Joules, which could be everything that the RAPL sensor is not able to measure like the cooling system and inefficiencies in the power supply. *PowerJoular* seems to only measure around half of what the *perf* meter calculates, which is surprising because it uses the exact same data source. The problem arises because of the used hardware of the tested server. It has two separate physical CPUs and *PowerJoular* does not support that [1, 41]. The energy consumption is therefore about half of the actual value. *Tracarbon* did not support multiple chips either, but during the development of CoMECS we opened a pull request [39] with a patch. This has already been merged and therefore the results of *Tracarbon* are not skewed in the same way the ones from *PowerJoular* are. The two variants of the *Cloud Energy* meter also show roughly the same result as *PowerJoular* even though they support multiple CPUs and have been configured via the command line arguments to assume two chips.

As for the results from the desktop shown in figures 4.5,4.6,4.7, it is difficult to interpret them, since the y-scale is so large, that the box plots are quite small. This is due to the *Cloud Energy* meters, which give very wrong results in both directions. This is not too surprising, since the SPECPower data set consists off server hardware and not desktop hardware. Figures 4.8,4.9,4.10 show the same data without these two meters. In these figures all the software meters have roughly the same result, except for *Tracarbon* in the memory tests. We can also see that for memory-focused benchmarks the software-based meters are off by only a small amount (15 Joules), for CPU-focused tests the difference is a bit bigger (100 Joules), but for the sequential read test, they only return about half of the real value. In absolute terms this is still only about 45 Joules, but it seems that none of the software meters track the real energy usage. More tests and potentially longer running tests are needed to get more conclusive results.

Overall however it seems like using *perf* (or *Tracarbon* and *CodeCarbon*) on server hardware and *CodeCarbon* or *perf* on consumer/desktop hardware gives the most accurate results in case a hardware meter is not available. Additionally the measuring techniques underlying the *Cloud Energy* project are not accurate for tests on bare metal hardware. This is not surprising, since it was designed for being used in CI-pipelines in Github without access to the *RAPL* sensor. This means it needs to model the energy consumption of the underlying hardware based on inaccurate data in a virtualized environment. Using

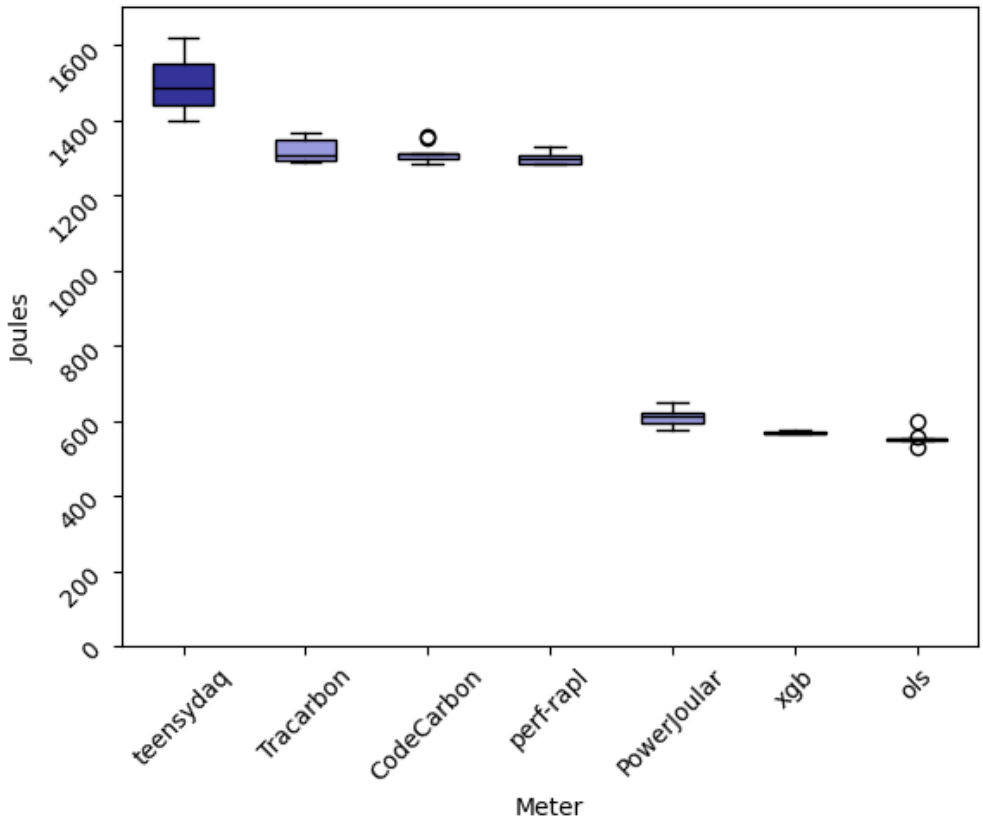


Figure 4.2: Power Consumption of sysbench cpu on the server

the same techniques on bare metal then probably introduces a lot of bias that is present in the training data.

4.2 Docker: PostgreSQL

Another interesting question that we wanted to answer is how much overhead virtualization causes. One very popular virtualization technique is containerization using Docker [10]. The overhead is often measured as performance overhead, which in most cases closely tracks with energy consumption. However we wanted to measure the energy consumption directly instead of assuming that it correlates with the performance overhead.

Our tests consisted of running the *pgbench* [55] benchmark against a PostgreSQL database that was running on the machine directly and one that was running in a Docker container. The command that we used to run the benchmark is `pgbench -T 30 -U postgres -h localhost postgres`. This command runs the test for 30s against a database via TCP connection to `localhost`. It does not limit the benchmark to a fixed number of transactions, but rather to a fixed amount of time. This allows to test how much energy

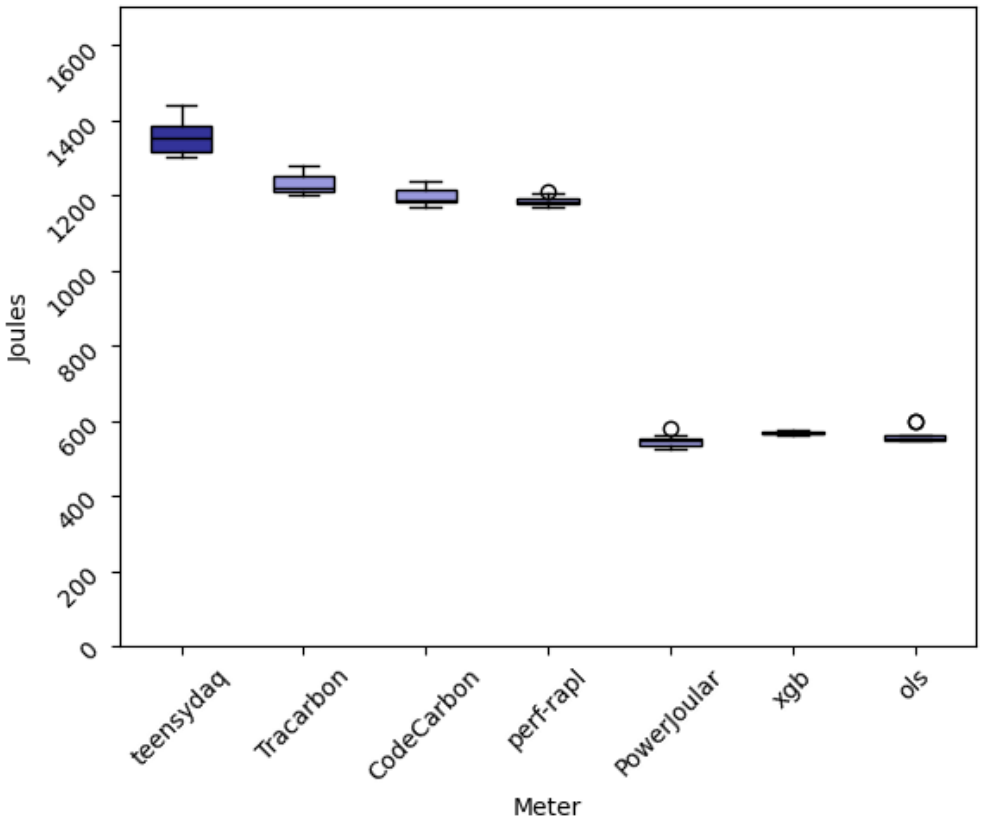


Figure 4.3: Power Consumption of sysbench memory on the server

is used during the same amount of time under the max load. It does not test much energy a single transaction would consume or how much efficiency overhead one version has over the other. This command is the same for the both versions, since both servers are configured to bind to `tcp://localhost:5432`. The baremetal database is started with the command `pg_ctl start -l $PGDATA/logfile -o "-p 5432"`, which starts the database listening on port 5432. The dockerized version uses the command `docker run -p5432:5432 -d postgres:16` to start the server, which uses Docker port mapping to expose the port 5432 from the container on the host system.

Our approach of testing the database system under max load tells us about how much the server consumes under worst case conditions. Another interesting metric is the energy efficiency for one transaction. This would consider the amount of transactions that are completed during the 30s test duration or measure how long n transactions take to execute. We have not tested this scenarios, but it is possible to run these kinds of experiment with CoMECS.

The results from these tests are shown in figure 4.11. As can be seen in the figures, on the server and the desktop the dockerized version of the database causes a measureable

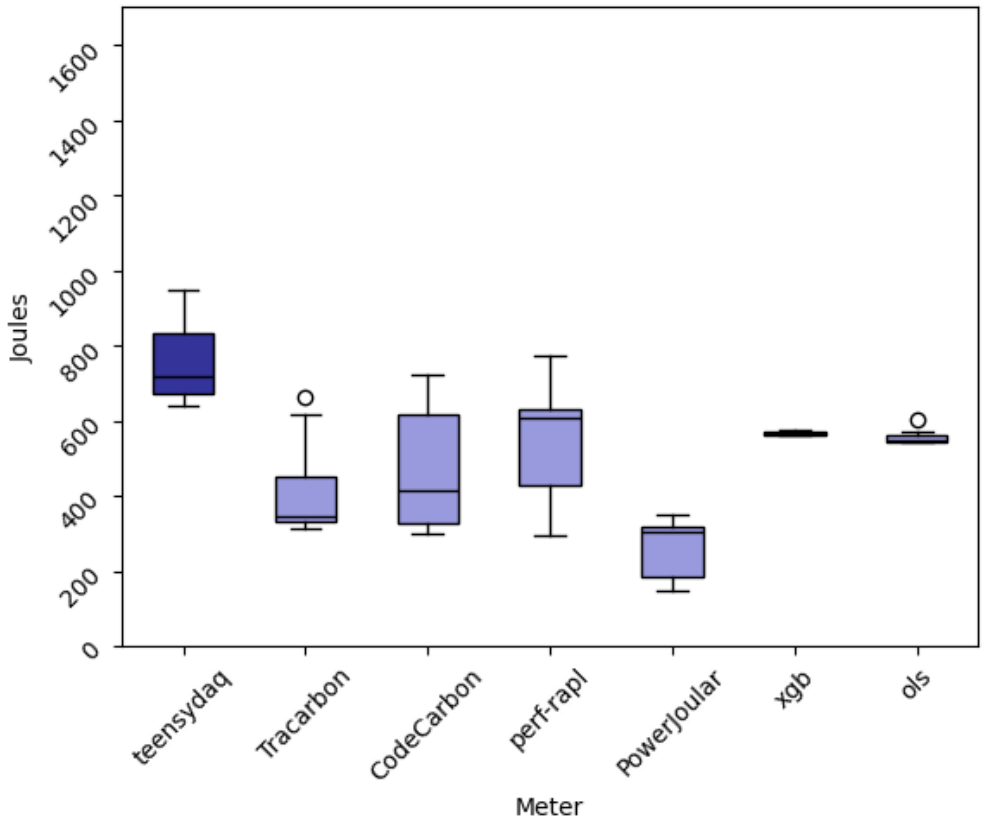


Figure 4.4: Power Consumption of sysbench io on the server

increase in the power consumption. On desktop hardware all the meters see a increase of the median energy consumption of around 10-25 Joules. For most of the meters the range of results also increases, which suggests that the variability increases. On the server hardware the results are similar, but the overall differences between the meters is much bigger. This higher variance could potentially be reduced by running the tests more often.

Previous research by Santos et al [49] also found that there is a statistically significant overhead for running software within Docker containers and their results also showed the same pattern of being more spread out using containerized software compared to bare metal software. Their hypothesis is that the Docker daemon that is running in the background and controlling the containers is causing the overhead.

These findings show that CoMECS is a tool capable for validating previous research, in addition to also being able to use multiple different meters as measuring backends.

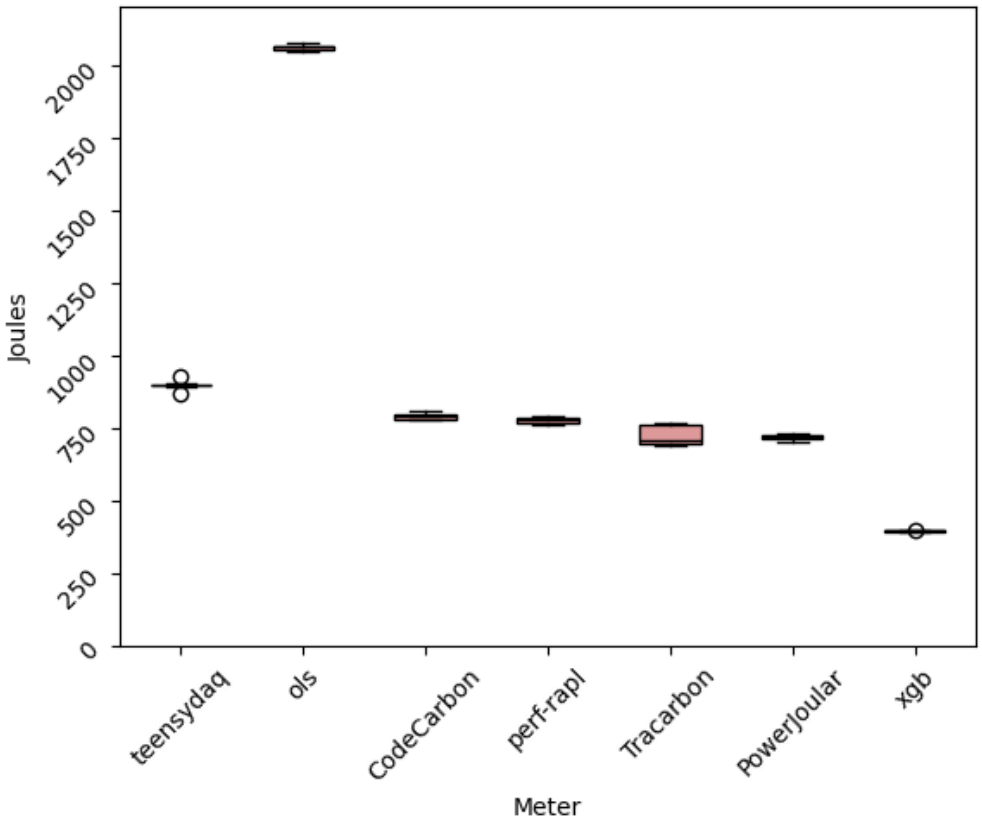


Figure 4.5: Power Consumption of sysbench cpu on the desktop

4.3 Parameterization: nginx

The question of how software changed over time in regards to its power consumption is very interesting. Answering this question can help track down changes that caused increased energy consumption. However such an experiment can be difficult to setup. We wanted to know if and how the popular webserver and reverse proxy *nginx* varied in its energy consumption over time. We decided to focus on the most recent releases of the versions `1.26` to `1.11`. These 16 different releases would need 16 different wrappers or 16 different make scripts that prepare all the dependencies and install the software, but the templating feature of CoMECS actually allowed us to write a single wrapper for all 16 versions with just one variable that sets the version. The wrapper includes one patch that is applied to the `ngx_user.c` file because of a changed API depending on the version. Other than that the wrapper has is not different to a wrapper for just a single specific version.

The experiment we conducted executes the *wrk* [61] benchmark against a running *nginx* instance with HTTPS enabled. The command that we defined for running the test is `wrk -d30s -c100 -t20 https://localhost:8123/`. The arguments define that the

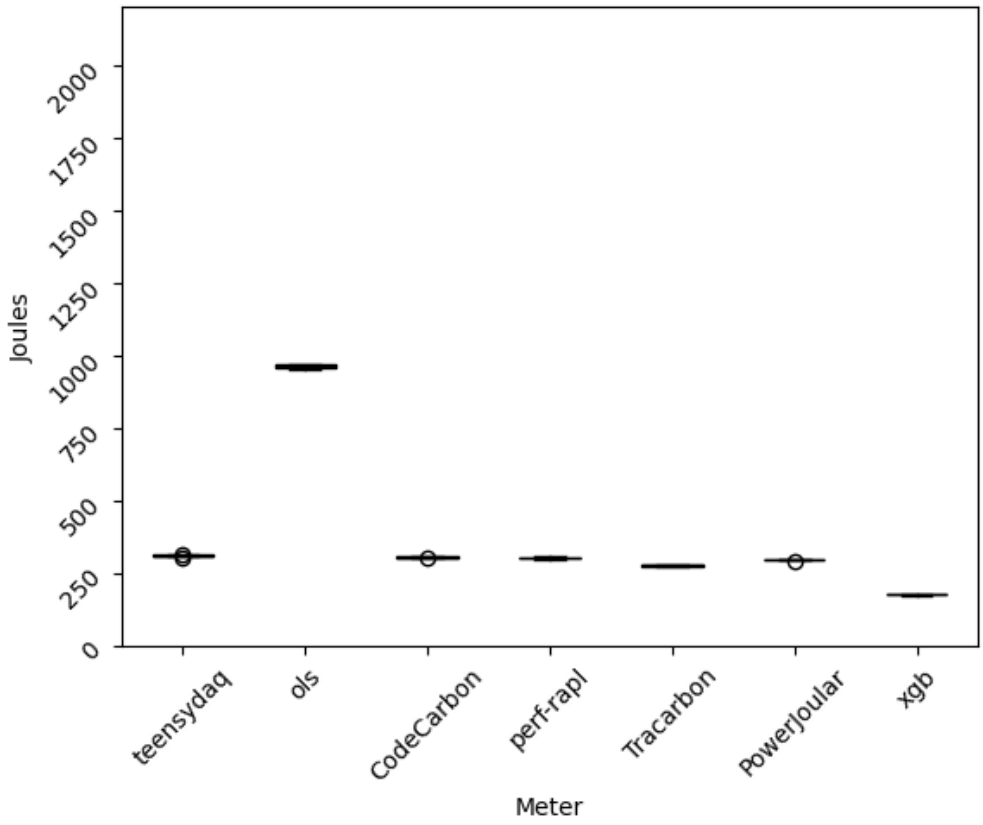


Figure 4.6: Power Consumption of sysbench memory on the desktop

benchmark should run for 30 seconds (`-d30s`) with 100 parallel connections (`-c100`) distributed over 20 threads (`-t20`). The started *wrk* instance then requests the URL `https://localhost:8123/` for the given duration as quickly as possible with the given constraints. For these tests we again chose the approach of benchmarking the server under max load instead of counting the requests made or limiting the test to n requests. The *nginx* web server was configured to enable HTTPS and with some unused routes that use regex matching, so that the path matching algorithm and the TLS encryption layer are also included when processing a request. The full configuration is listed in appendix 6.2. The configuration is based on the one [30] used by the Phoronix Test Suite [43].

This experiment is executed 10 times for every defined nginx version, which results in a total of 160 test runs. The results are shown in figure 4.12. As can be seen in the figure the tests have been executed only using the hardware meter, due to the long running nature of the experiment and because it is not obvious or immediately clear which type of resource, i.e. CPU, main memory or io, would be the limiting factor for this benchmark. Since only some software meters yield satisfying results for certain workloads. Therefore

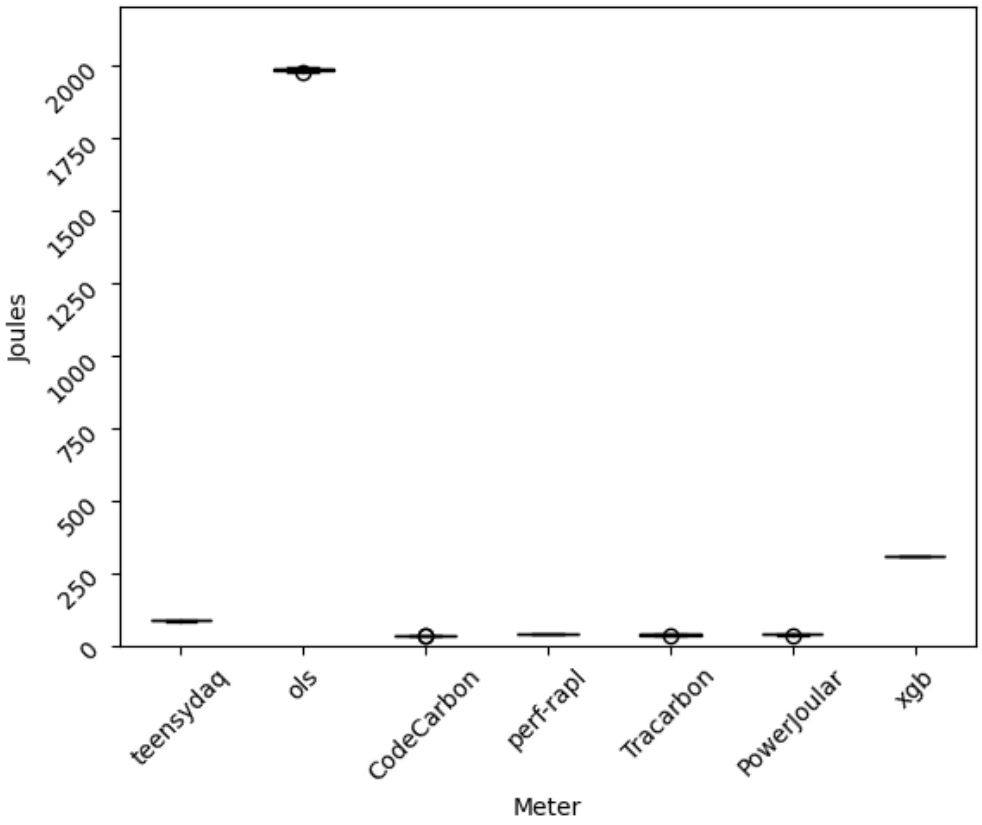


Figure 4.7: Power Consumption of sysbench io on the desktop

we decided to only use the hardware meter as it would offer the most reliable results for this test. The graphs for desktop and server both show a downward trend for the average of the consumed power. The Pearson coefficient for the averages over time is -0.64 and -0.76 for the server and desktop respectively. This indicates a strong correlation between the energy consumption and the versions of nginx. The absolute difference of average energy consumption between the first and last tested version is only around 100 Joules, which is only 1-3% but multiplied by the amount of deployed nginx instances, this amounts to a lot of saved energy. When we assume the number of nginx instances to be 200 million (see 1.2), then this amounts to around

$$\begin{aligned} 200 * 10^6 \text{ devices} * 100J = \\ \frac{20 \text{ GJ}}{30 \text{ s}} = \\ \sim 670 \text{ MJ} \end{aligned}$$

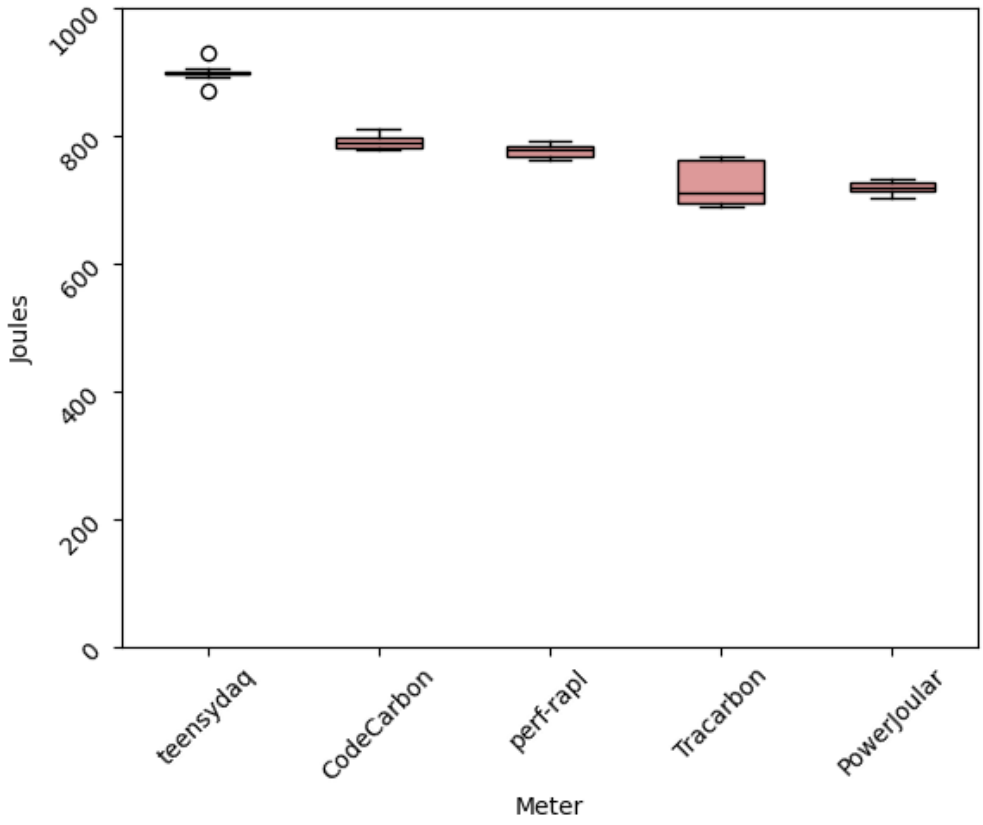


Figure 4.8: Power Consumption of sysbench cpu on the desktop

of conserved energy per second, which is

$$670 * 10^6 \text{ J} * 60\frac{\text{s}}{\text{min}} * 60\frac{\text{min}}{\text{h}} * 24\frac{\text{h}}{\text{day}} * 365 \text{ days} * \frac{1 \text{ Wh}}{3600 \text{ J}} = 5869.2 * 10^9 \text{ Wh} = \sim 6 \text{ TWh}$$

in one year. With a global energy consumption of 415 TWh for data centers [20], this reduction alone saves around 1.5%. This of course only makes sense under the assumption that every single deployment runs similar hardware, switches from version 1.11 to 1.26 and has similar kinds of workloads than we simulated in our benchmarks. So the result is the best case scenario. The impact in the real world is most likely smaller.

The diagrams also show a sudden decrease in energy consumption for version *1.19.10*. The change that we think might have caused this drop is that the `keepalive_requests` default value was increased from 100 to 1000 [29]. The nginx documentation describes the value as *...the maximum number of requests that can be served through one keep-alive connection. After the maximum number of requests are made, the connection is*

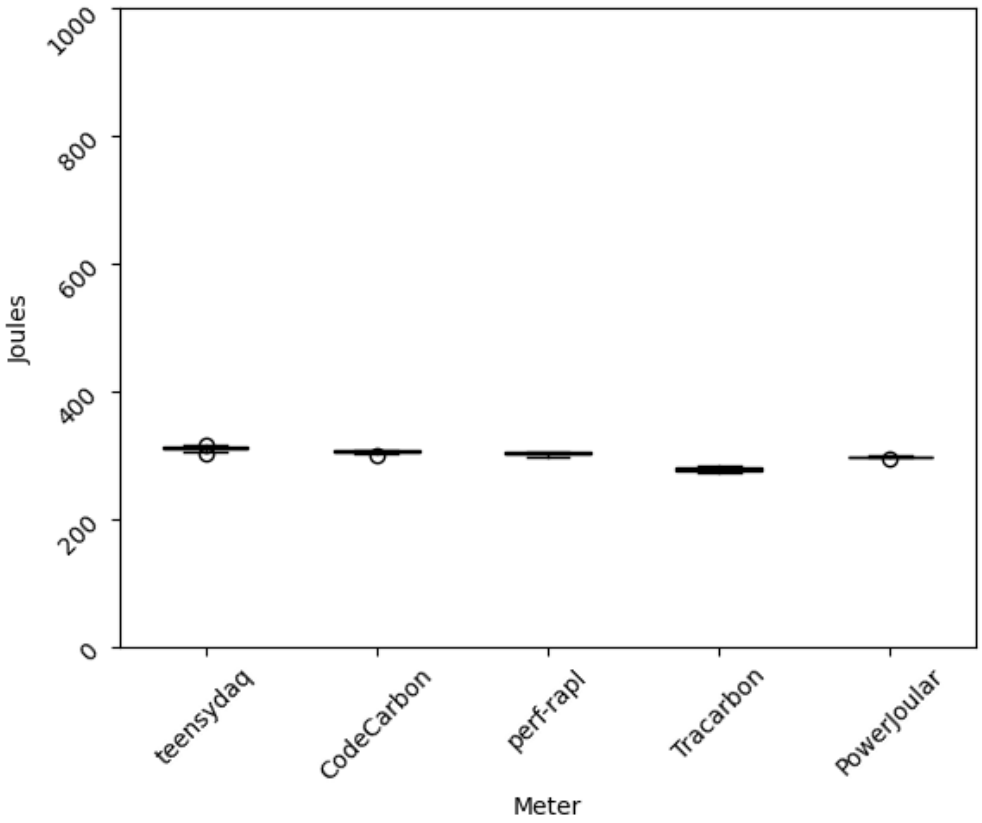


Figure 4.9: Power Consumption of sysbench memory on the desktop

closed. [32]. In our case this would mean that for versions before 1.19.10 after at most 100 requests a connection was forcibly closed and reopened, which uses more resources than reusing the already opened connection. Since we are running the test under max load with 100 concurrent requests with 20 threads, this would happen fairly quickly and often. However this is just our hypothesis and would need to be verified by setting the value explicitly so that earlier version also use the value of 1000 or later versions the value 100. If the drop is not present anymore, then this would be a strong indicator that our hypothesis is true.

The results from this experiment show how the parameterization in addition to the declarative nature of the wrappers facilitate experiments that would usually take much more effort to set up.

4.4 Artificial Intelligence and GPUs

The report by the International Energy Agency [20] predicts that data centers will more than double until 2030. The rise in the popularity of artificial intelligence (AI) in recent

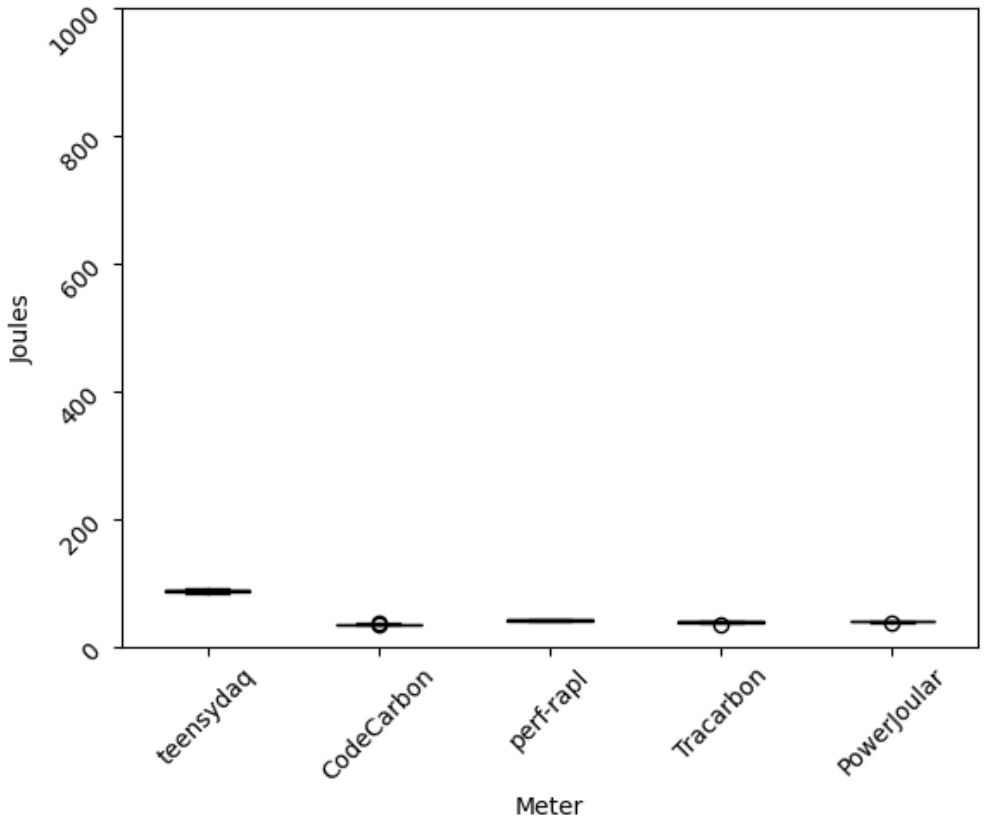
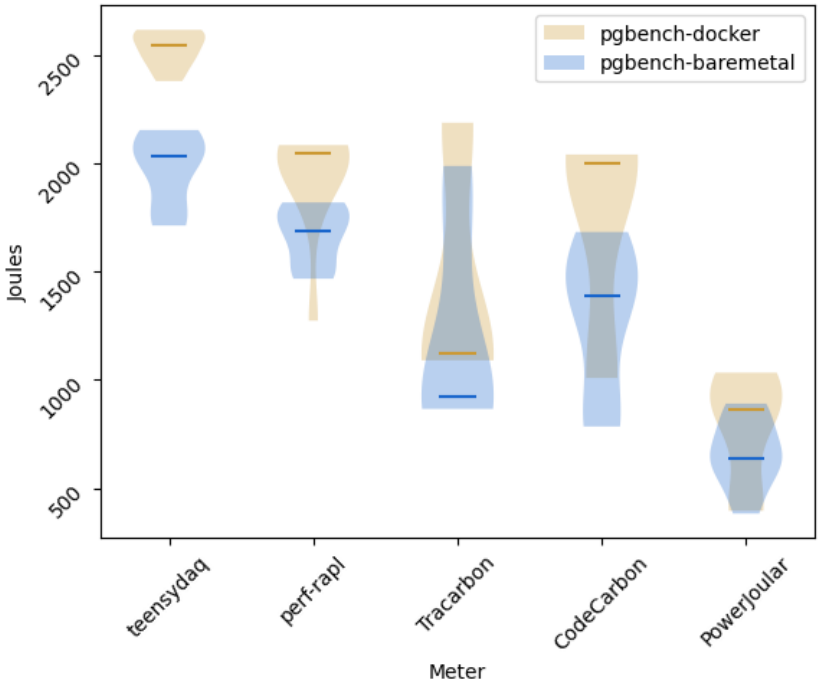


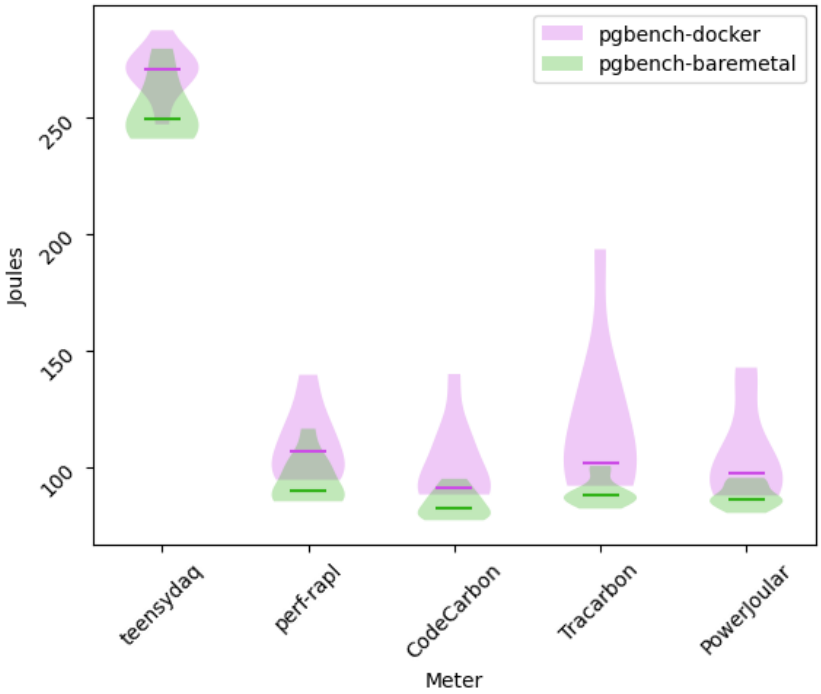
Figure 4.10: Power Consumption of sysbench io on the desktop

years is the driver behind this trend. AI includes everything from linear regression models to diffusion models for image generation, but the most important right now are transformer models, also called *Large Language Models* or *LLM*s. These models are huge and therefore need a lot of energy. The first public breakthrough was the release of *ChatGPT* in 2022 [38]. Since then *LLM*s have gained popularity and a lot of other companies started training and releasing these models to the world. The main problem with these products is that their training and execution is extremely resource intensive. During training server farms filled with the newest, biggest and most energy intensive GPUs are running for months to train a single model. This is the main reason behind the prediction of the International Energy Agency. These heavy GPU workloads are therefore an interesting target for measuring their energy consumption.

The machines we had access to for our experiments however, have no GPUs. This would have allowed us to test these kinds of workloads. From the information we gathered, it is however clear that the RAPL sensor, which we proposed as being one of the best choices for estimating the real energy consumption, is not capable of measuring the energy usage of an external GPUs. The *Tracarbon* and *CodeCarbon* tools allow to measure the power

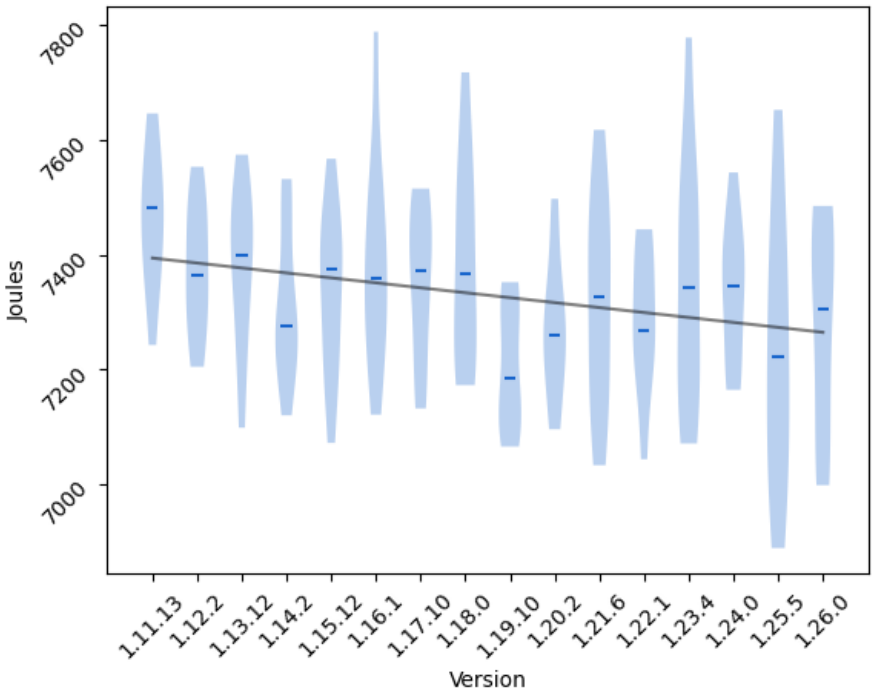


(a) Server

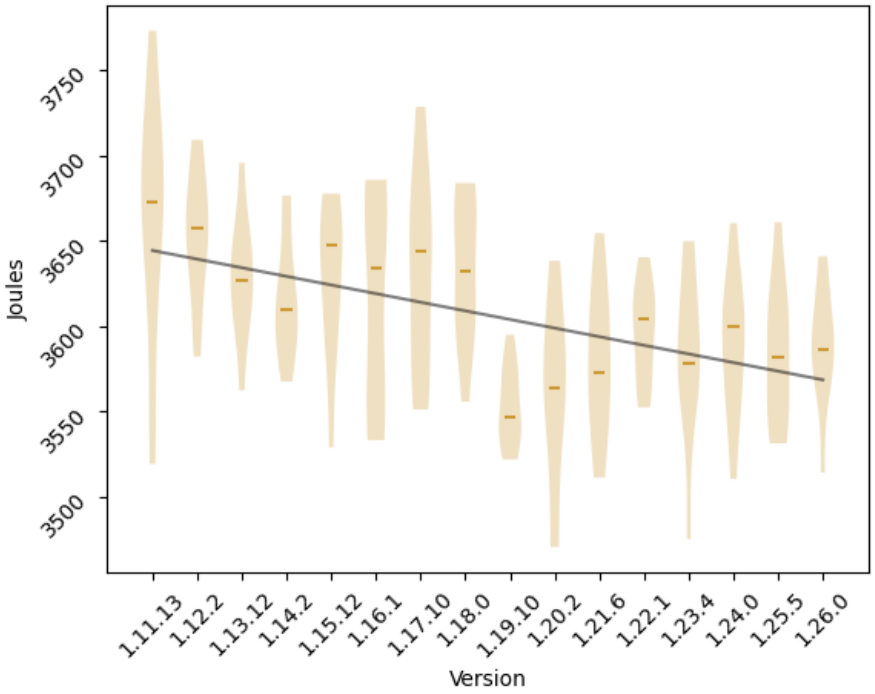


(b) Desktop

Figure 4.11: Power Consumption of the PostgreSQL benchmark



(a) Server



(b) Desktop

Figure 4.12: Energy Consumption of the nginx

consumption of NVIDIA GPUs by using the *NVIDIA Management Library* [36]. So these two tools might be the better choice for measuring software that rely on computations on the GPU through the computing framework CUDA [37] or graphic APIs like OpenGL [53] and Vulkan [54]. It is however not clear if the setups used in the datacenters specialized in training AI models would be covered by the capabilities of *Tracarbon* and *CodeCarbon*.

4.5 Conclusion & Future Research

The goals set at the begin of chapter 3 where **Openness**, **Reproducibility** and **Standardized Output**. All the tests ran on two very different machines without needing any change in the code of the framework or the wrappers. The tests have been executed fully automatically. The only human interaction needed was starting the experiment by executing a single command on the terminal. After the execution of the experiments all results have been stored to the results directory. The files needed no additional work and could be directly used to render the diagrams. The output of the tests all have the exact same CSV format. There is no difference between measuring the energy consumption of *pgbench* on *PostgreSQL* with *perf* on the desktop PC or of *nginx* version *1.19.10* with the hardware meter on the server. This made it possible to easily render all the diagrams and compare the results without having to rely on expensive manual transformation effort. After running the previously described experiments and evaluating the results, we conclude that the defined goals have been achieved by the implemented framework.

The conducted experiments and their results show that CoMECS can be used for wide range of different research questions regarding energy consumption of programs thanks to the powerful wrapper and parameterization system. We argued that available open-source based power meter software is useful for CPU and memory focused tasks, but needs improvement for workloads that are primarily reading and writing to disk. Also, for hardware with multiple physical CPUs the program needs to be chosen carefully, since not all of them support multiple chips. The experiments showed that running software in a containerized environment with Docker has a no insignificant energy overhead. Further research is needed to determine the exact cause for this, but a previous paper by Santos et al suggests that the main cause is the Docker daemon that is running the background. Experiments with daemon-less alternatives like Podman [44] could clarify if this is the case or if the thin-wrapper around the kernel, which is the basis of containerization, is the main driver of the increased energy use. The results from the test on the nginx web server hint towards a downward trend in the energy consumption of the software, but this needs further research and more in-depth analysis to conclude this with certainty. There is also an interesting drop in the energy usage with version *1.19.10*, which would be interesting to look into.

In the future CoMECS could be used to improve existing power meters. It would also facilitate the creation of new ones, by allowing to easily compare them to already existing ones, which is really important to help researchers and developers. The framework could also be used to establish a long running benchmark that measures software regularly and

publishes the results. An instance could also be hosted and made available to the public to conduct their own research. Since the framework is written in a quite general way, it could also proof useful for measuring more than just energy consumption. The only part of the system that assumes a very strict format is the computation of the results by subtracting the baseline measurement from the test results, but this can be easily removed or adapted. This would open up the possibilities to capture different metrics like CPU utilization, main memory usage, disk IO speeds and much more, which could proof to be useful for applications beyond the initial scope of the framework. The source code of CoMECS will be made available publicly as an open-source project, so that the results from this thesis can be validated and researchers can build on top of this framework for future research.

The CoMECS framework proofed to be a useful tool for measuring the energy consumption and comparing the available solutions. It might not revolutionize the IT world and solve climate change, but it adds something to the toolkit of future software developers, engineers and researchers, which might facilitate the reduction of the energy consumption of software. It is only a tiny improvement, however it is a step in the right direction on the long road ahead.

CHAPTER 5

Conclusion

Motivation & Background

The human-made climate change is the biggest challenge that we as a species have to face. It is the ultimate cross-cutting concern that reaches every single part of our lives. Each scientific field and engineering discipline needs to contribute to a solution. The various international agreements and also more localized efforts like the European Green Deal, show that this challenge is generally taken seriously and that a solution is desired. Computer science also has to contribute its part. Not only because it is part of every major industry, or because nearly every person uses a smartphone or computer daily, but because the energy consumption of the information technology sector is predicted to more than double between 2025 and 2030 by the International Energy Agency [20].

Computer science and engineering has a long history of increasing performance and finding efficient solutions. These usually focus on computing the result quicker, which can decrease the energy consumption. This is however not a given. Therefore the research and engineering community started creating tools to measure energy consumption directly. We described some of the available tools. Namely the *RAPL* sensor, *perf*, *PowerJoular*, *CodeCarbon*, *Tracarbon* and *Eco-CI* (and the underlying *Cloud Energy* project). These tools allow to measure the energy consumption of executed programs. There is just one major challenge when using them - their usage and output is very different from each other. *RAPL* can be read via CPU registers or mapped files on Linux systems and the *Eco-CI* project is made for *Continuous Integration* pipelines in *GitHub*. Comparing them is challenging and requires knowledge about each one of them.

Framework Goals

Unifying all of these tools and a hardware meter under a single command line interface was the goal set for this thesis. The various tools should be reproducibly executable on

different systems without needing to fine tune parameters or configuration. The output should have the same format no matter which test is executed or which meter used for the measurement and therefore comparable. The test machines should also be as bare-bones as possible. Every additional software running the in the background could skew the measuring results. Therefore the communication with the measured system should use standard protocols like SSH and nothing else.

Implementation

The implementation of this framework had to overcome some challenges, like making it possible for different measuring tools that are not limited to the ones we decided on for this thesis. One of the challenges was overcoming the limitations of SSH. The protocol is meant for human interaction or single automated commands. What our framework needs however is to automate many commands as if they were coming from a human, which includes opening shells within shells. Our solution for this problem was to use the SSH connection as if it was a raw TCP socket connection, which sends and receives bytes without synchronization mechanism. We then added markers around each command. This approach allowed us wait until a command has finished and receive its results in a synchronized way. Even when we spawned shells within shells, which would usually not be possible with most SSH client libraries. Another question to answer was how to wrap vastly different tools into the same framework. We came up with a wrapping technique based on a nix technique called nix-shell. These are easy to write declarative descriptions of what packages one needs. In addition they cause minimal overhead on the measured system. Since they are based on nix, there is already a huge amount of packages that are readily available and it can be installed on every major distribution. Additionally we used Python's string templates to add even more expressiveness into the framework. The templates are very simple, since they only allow placeholders for variables, which are then filled in with actual values. When combining these templates with parameters that are declarable via command line arguments or a configuration file, this leads to a very expressive system, which reduces duplication without being too complicated. This allows to define complex experiments, like measuring the energy consumption of *nginx* through various versions with a single wrapper. These kinds of experiments would usually need a lot of manual work to setup, but with our parameterization system it can be as simple as defining a single variable and adding placeholders into a script.

Evaluation

After the implementation, we evaluated the framework in three different experiments. The first one compared the software-based power meters with a hardware-based one. We found that *perf*, *CodeCarbon* and *Tracarbon* give the most accurate results, when compared against ourthe hardware meter for CPU and memory heavy workloads. However for applications that mostly do disk reads and writes, they might not capture the actual energy consumption. In the second experiment we verified previous results about the

overhead of containerization using Docker. In particular with the *PostgreSQL* database system. The results of our measurements showed an overhead when using Docker, which are similar to previous research results. The last experiment relied on the parameterization capabilities of the framework to measure 16 different versions of the *nginx* webserver. We found that between versions 1.18 and 1.19 there was a significant improvement in energy consumption. Over the span of all the versions there is also a downward trend in the electricity usage of nginx.

Outlook

The results from the evaluation step shows that the framework is achieving the previously defined goals. It was used for validating previous results regarding the overhead of containerization on the energy consumption. Different power meters have been compared to each other and a hardware meter acting as the baseline. Additionally the experiment on the energy consumption for nginx over its version history showed how powerful the framework is and how it can be used for previously difficult-to-setup experiments.

The CoMECS framework was published as open source project [40] under the MIT license. This allows researchers to use it and build on top of it. One usage would be to evaluate a newly developed power meter by comparing it to already available ones. Another way to use this research would be to use it for measuring the energy consumption of a software project regularly or for the computation of the SCI. This would allow engineers and developers to define goals for reducing the energy consumption of their software. Our hope with CoMECS is that it will have at least a small positive impact in the mission to cut emissions in the IT sector.

CHAPTER 6

Appendix

6.1 Wrapper Examples

6.1.1 Test

File `config.toml`:

```
name = "Numpy Matrix Multiplication"
description = "This tests how much energy is used for matrix
   multiplications in numpy"
system = "Python 3.12"
```

File `shell.nix`:

```
{ pkgs ? import <nixpkgs> {} } :
pkgs.mkShell {
      nativeBuildInputs = with pkgs [
            python312Packages.numpy
      ];
}
```

File `test.py.template`:

```
#!/usr/bin/env python3

import numpy as np

matrix1 = np.random.rand($matrix_size, $matrix_size)
matrix2 = np.random.rand($matrix_size, $matrix_size)

for i in range($runs):
      np.matmul(matrix1, matrix2)
```

The template variables `$matrix_size` and `$runs` have to be provided through command line arguments or the configuration file.

6.1.2 System

File `config.toml`:

```
name = "Python 3.12"
description = "Python release 3.12"
```

File `shell.nix`:

```
{ pkgs ? import <nixpkgs> {} } :
pkgs.mkShell {
      nativeBuildInputs = with pkgs [
            python312
      ];
}
```

6.1.3 Meter

File `config.toml`:

```
[general]
name = "perf-rapl"
description = "perf is a Linux tool that allows to use performance
   counter subsystems in Linux. This software meter uses the power/
   energy-psys/ event to get the power usage of the whole system"
url = "https://perf.wiki.kernel.org/index.php/Main_Page"
converter = true
```

File `shell.nix`:

```
{ pkgs ? import <nixpkgs> {} } : pkgs.mkShell {
      nativeBuildInputs = with pkgs.buildPackages; [
            linuxPackages_latest.perf
      ];
}
```

File `converter`:

```
#!/usr/bin/env python3

import sys
import re

input_file = sys.argv[1]
output_file = sys.argv[2]

with open(input_file, "r") as f:
```

```
        input_content = f.read()

line_regex = r"\s+(\d+\.\d+)\s+(\d+[,\.]\d+)\s+"

tick = 0
output_content = "Tick;Joules"
for line in input_content.split("\n"):
        if line.strip().startswith("#") or len(line.strip()) == 0:
                continue

        _, joules = re.match(line_regex, line).group(1, 2)
        joules = float(joules.replace(",", "."))

        output_content += f"\n{tick};{joules}"
        tick += 1

with open(output_file, "w") as f:
        f.write(output_content)
```

File `wrapper`:

```
#! /usr/bin/env bash
perf stat -a -e "power/energy-pkg/" -I 1000 2> $1
```

6.2 nginx Configuration

```
worker_processes auto;

events {
        worker_connections 1024;
}

http {
        include    mime.types;
        default_type application/octet-stream;

        sendfile    on;
        keepalive_timeout 65;

        server {
                listen    8123 ssl;
                server_name localhost;

                ssl_certificate ../certs/localhost.cert;
                ssl_certificate_key ../certs/localhost.key;
                ssl_ciphers HIGH:!aNULL:!MD5;

                add_header X-Robots-Tag "noindex";
```

```
                location / {
                      root html;
                      index index.html index.htm;
                }

                location ~* "\.(old|orig|original|php#|php~|php_bak|save|
                   swo|aspx?|tpl|sh|bash|bak?|cfg|cgi|dll|exe|git|hg|ini|
                   jsp|log|mdb|out|sql|svn|swp|tar|rdf)$" {
                      deny all;
                }

                location ~ /\.(?!well-known\/) {
                      deny all;
                }

                location ~* "(base64_encode)(.*)(\()" {
                      deny all;
                }

                location ~* "(eval\()" {
                      deny all;
                }

                error_page 500 502 503 504 /50x.html;
                location = /50x.html {
                      root html;
                }
        }
}
```

6.3 Configuration files JSON schema

6.3.1 Framework - config.toml

```
{
      "type": "object",
      "properties": {
            "paths": {
                  "type": "object",
                  "properties": {
                        "tests": {
                              "type": "string"
                        },
                        "systems": {
                              "type": "string"
                        },
                        "meters": {
```

```
                        "type": "string"
                    },
                    "output": {
                        "type": "string"
                    }
                },
                "additionalProperties": false
            },
            "connection": {
                "type": "object",
                "properties": {
                    "host": {
                        "type": "string"
                    },
                    "port": {
                        "type": "integer",
                        "minimum": 0,
                        "maximum": 65535
                    },
                    "username": {
                        "type": "string"
                    },
                    "password": {
                        "type": "string"
                    }
                },
                "additionalProperties": false
            },
            "parameters": {
                "type": "object",
                "additionalProperties": {
                    "type": "array"
                }
            },
            "runner": {
                "type": "object",
                "properties": {
                    "user": {
                        "type": "object",
                        "properties": {
                            "name": {
                                "type": "string"
                            },
                            "uid": {
                                "type": "integer",
                                "minimum": 0
                            },
                            "gid": {
                                "type": "integer",
```

```
                                                "minimum": 0
                                          }
                                    },
                                    "additionalProperties": false
                              },
                              "runs": {
                                    "type": "integer",
                                    "minimum": 1
                              }
                        },
                        "additionalProperties": false
                  }
            },
            "additionalProperties": false
}
```

6.3.2 Meter Wrapper - config.toml

```
{
      "type": "object",
      "properties": {
            "general": {
                  "type": "object",
                  "properties": {
                        "name": {"type": "string"},
                        "description": {"type": "string"},
                        "converter": {"type": "boolean"},
                        "url": {"type": "string"},
                        "calibration-time": {"type": "integer"},
                        "variables": {"type": "array", "items": {"
                           type": "string"}},
                  },
                  "additionalProperties": False,
                  "required": ["name", "description", "converter"],
            },
      },
      "additionalProperties": False,
      "required": ["general"],
}
```

6.3.3 System Wrapper - config.toml

```
{
      "type": "object",
      "properties": {
            "general": {
                  "type": "object",
                  "properties": {
```

```
                        "name": {"type": "string"},
                        "description": {"type": "string"},
                        "variables": {"type": "array", "items": {"
                            type": "string"}},
                    },
                    "additionalProperties": False,
                    "required": ["name", "description"],
                },
        },
        "additionalProperties": False,
        "required": ["general"],
}
```

6.3.4 Test Wrapper - config.toml

```
{
        "type": "object",
        "properties": {
                "general": {
                    "type": "object",
                    "properties": {
                        "name": {"type": "string"},
                        "description": {"type": "string"},
                        "system": {"type": "string"},
                        "command": {"type": "string"},
                        "variables": {"type": "array", "items": {"
                            type": "string"}},
                    },
                    "additionalProperties": False,
                    "required": ["name", "description", "system", "
                        command"],
                },
        },
        "additionalProperties": False,
        "required": ["general"],
}
```

Overview of Generative AI Tools Used

None

Übersicht verwendeter Hilfsmittel

Keine

List of Figures

2.1 Overview of the setup using the hardware meter 9
2.2 State machine of the hardware meter 9
2.3 C-struct data format for data sent by the hardware meter 10
2.4 RAPL Architecture [27] 11
2.5 Pseudo-code for retrieving the current energy consumption from the RAPL sensor on Linux 12

3.1 Experiment process overview 19
3.2 Test Setup 20
3.3 Test setup with an additional measurement machine 21
3.4 Wrapper for a synchronous shell command 23
3.5 Wrapper for activating an environment 24
3.6 Wrapper Structures 25
3.7 `config.toml` file for the *Tracarbon* meter 25
3.8 `config.toml` file for the test, which executes *pgbench* 26
3.9 `shell.nix` file for a custom *nginx* package 26
3.10 Computation of parameter configurations 28
3.11 Test Process 31
3.12 Test Result File Structure 31
3.13 Calculation of the amount of data the hardware meter sends 34

4.1 Hardware Specifications 37
4.2 Power Consumption of sysbench cpu on the server 41
4.3 Power Consumption of sysbench memory on the server 42
4.4 Power Consumption of sysbench io on the server 43
4.5 Power Consumption of sysbench cpu on the desktop 44
4.6 Power Consumption of sysbench memory on the desktop 45
4.7 Power Consumption of sysbench io on the desktop 46
4.8 Power Consumption of sysbench cpu on the desktop 47
4.9 Power Consumption of sysbench memory on the desktop 48
4.10 Power Consumption of sysbench io on the desktop 49
4.11 Power Consumption of the PostgreSQL benchmark 50
4.12 Energy Consumption of the nginx 51

Bibliography

[1] Adel Noureddine. PowerJoular code comment. https://github.com/joular/powerjoular/blob/4bc2579c78d577a69d1d59adbcead06d58deaa4e/src/intel_rapl_sysfs.ads#L43.

[2] Alexey Kopytov. sysbench. https://github.com/akopytov/sysbench.

[3] Analog Devices, Inc. ADE9153A - Energy Metering IC with Autocalibration. https://www.analog.com/en/products/ade9153a.html.

[4] ansible. ansible. https://github.com/ansible/ansible.

[5] Arjan van de Ven. PowerTOP. https://github.com/fenrus75/powertop.

[6] BCG Gamma, Comet.ml, Haverford College, MILA. CodeCarbon: Track and reduce CO2 emissions from you computing. https://github.com/mlco2/codecarbon, 2023.

[7] BuiltWith Pty Ltd. Web Server Usage Distribution in the Top 1 Million Sites. https://trends.builtwith.com/web-server.

[8] S. Böck. *Erfassung und Beurteilung des Energieverbrauchs von Maßnahmen im Bereich der IT-Security.* Masterarbeit, FH Technikum Wien, Wien, Apr. 2021.

[9] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, page 189–194, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Docker Inc. Docker. https://www.docker.com/.

[11] European Commission, Secretariat-General. Communication on The European Green Deal. https://commission.europa.eu/document/daef3e5c-a456-4fbb-a067-8f1cbe8d9c78_en, 2019.

[12] F5, Inc. NGINX. https://www.nginx.com/.

[13] Florian Valeye. Tracarbon. https://github.com/fvaleye/tracarbon.

[14] G. S. Foundation. Software Carbon Intensity (SCI) Specification. https://github.com/Green-Software-Foundation/sci/blob/main/SPEC.md, 2022.

[15] Github - JulienMalka. Commit history nixos-.../pkgs/.../powerjoular/package.nix. https://github.com/NixOS/nixpkgs/commits/nixos-24.11/pkgs/by-name/po/powerjoular/package.nix.

[16] Google. Go. https://go.dev/.

[17] Green Coding Berlin. Eco CI Energy estimation for Github Actions Runner VMs. https://github.com/green-coding-berlin/eco-ci-energy-estimation, 2023.

[18] Green Coding Solutions. Cloud Energy. https://github.com/green-coding-solutions/cloud-energy.

[19] hukkin. tomli. https://github.com/hukkin/tomli.

[20] IEA. Energy and AI. https://www.iea.org/reports/energy-and-ai.

[21] Intel®. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, 3C, and 3D: System Programming Guide. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2023.

[22] InternetLiveStats.com. Total number of Websites. https://www.internetlivestats.com/total-number-of-websites/.

[23] IPCC. Summary for Policymakers. In: Climate Change 2023: Synthesis Report.Contribution of Working Groups I, II and III to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change [Core Writing Team, H. Lee and J. Romero (eds.)]., 2023.

[24] Jackson Roland. Eunice Foote, John Tyndall and a question of priority, 2020.

[25] Jake Edge. Perfcounters added to the mainline. https://lwn.net/Articles/339361/.

[26] JSON Schema. JSON Schema. https://json-schema.org/.

[27] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou. RAPL in Action: Experiences in Using RAPL for Power Measurements: ACM Transactions on Modeling and Performance Evaluation of Computing Systems: Vol 3, No 2, Mar. 2018.

[28] J. Malmodin and D. Lundén. The Energy and Carbon Footprint of the Global ICT and E&M Sectors 2010–2015. 10(9):3027, 2018.

[29] Maxim Dounin. nginx - Commit eb52de8. https://github.com/nginx/nginx/commit/eb52de83114e8d98722cd17ec8435c47956b6315.

[30] Michael Larabel. nginx Benchmark for the Phoronix Test Suite. https://github.com/phoronix-test-suite/test-profiles/tree/master/pts/nginx-3.0.1.

[31] Netcraft Ltd. December 2023 Web Server Survey. https://www.netcraft.com/blog/december-2023-web-server-survey/.

[32] nginx. Module ngx_http_core_module. https://nginx.org/en/docs/http/ngx_http_core_module.html#keepalive_requests.

[33] Nix Documentation Team. Nix. https://nix.dev/.

[34] Nix Documentation Team. Nix Shell. https://nix.dev/tutorials/first-steps/declarative-shell.html.

[35] A. Noureddine. Powerjoular and joularjx: Multi-platform software power monitoring tools. In *18th International Conference on Intelligent Environments (IE2022)*, Biarritz, France, Jun 2022.

[36] NVIDIA Corporation . NVIDIA Management Library (NVML). https://developer.nvidia.com/management-library-nvml.

[37] Nvidia Corporation. Cuda Toolkit. https://developer.nvidia.com/cuda-toolkit.

[38] OpenAi. Introducing ChatGPT. https://openai.com/index/chatgpt/.

[39] Patric Gruber. Add support for multiple packages for RAPL sensor. https://github.com/fvaleye/tracarbon/pull/392.

[40] Patric Gruber. CoMECS - Source Code Repository. https://github.com/patgruber/comecs.

[41] Patric Gruber. Github issue: Add support for hardware configurations with multiple packages. https://github.com/joular/powerjoular/issues/81.

[42] Perf Wiki. perf. https://perf.wiki.kernel.org/index.php/Main_Page.

[43] Phoronix Media. Phoronix Test Suite - Open-Source, Automated Benchmarking. https://www.phoronix-test-suite.com/.

[44] Podman. podman. https://podman.io/.

[45] Python. Template strings. https://docs.python.org/3/library/string.html#template-strings.

[46] Python Packaging Authority. The Python package installer. https://github.com/pypa/pip.

[47] Python Software Foundation. argparse — Parser for command-line options, arguments and subcommands. https://docs.python.org/3/library/argparse.html.

[48] G. Raffin and D. Trystram. Dissecting the software-based measurement of cpu energy consumption: A comparative analysis. *IEEE Transactions on Parallel and Distributed Systems*, 36(1):96–107, 2025.

[49] E. A. Santos, C. McLean, C. Solinas, and A. Hindle. How does Docker affect energy consumption? Evaluating workloads in and out of Docker containers, May 2017. arXiv:1705.01176 [cs].

[50] SimilarTech. Total Web Server Technologies. https://www.similartech.com/categories/web-server.

[51] Siteefy. How Many Websites Are There in the World. https://siteefy.com/how-many-websites-are-there/.

[52] Standard Performance Evaluation Corporation. All SPECpower_ssj2008 Results. https://spec.org/power_ssj2008/results/power_ssj2008/.

[53] The Khronos® Group Inc. OpenGL Website. https://www.opengl.org/.

[54] The Khronos® Group Inc. Vulkan Website. https://www.vulkan.org/.

[55] The PostgreSQL Global Development Group. pgbench - Benchmark test for PostgreSQL. https://www.postgresql.org/docs/current/pgbench.html.

[56] UNFCCC. The Rio Conventions. https://unfccc.int/process-and-meetings/the-rio-conventions, 1992.

[57] UNFCCC. Kyoto Protocol to the United Nations Framework Convention on Climate Change. https://unfccc.int/documents/2409, 1997.

[58] UNFCCC. Paris Agreement. https://unfccc.int/documents/37107, 2015.

[59] United Nations. United Nations Framework Convention on Climate Change. https://unfccc.int/.

[60] Wappalyzer. Web servers. https://www.wappalyzer.com/technologies/web-servers/.

[61] Will Glozer. Modern HTTP benchmarking tool. https://github.com/wg/wrk.