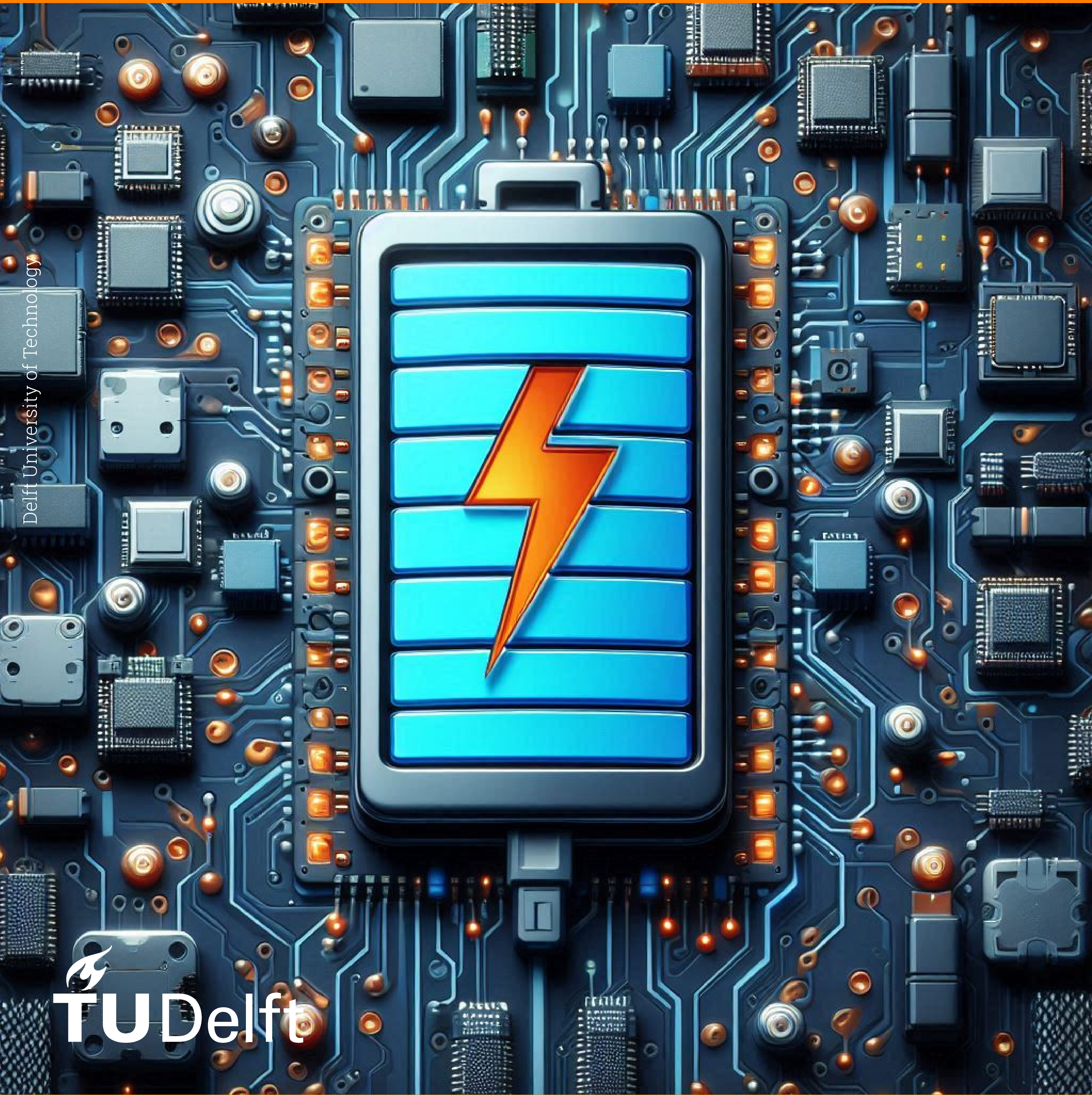
Exploration of the power of software

Bertrand Leclercq



Delft University of Technology

Exploration of the power of software

by

Bertrand Leclercq

to obtain the degree of Master of Science
in Embedded Systems
at the Delft University of Technology,
to be defended publicly on Tuesday July 2, 2024 at 10:30 AM.

Student number:	4869966	
Project duration:	March 2023 – June 2024	
Thesis committee:	Dr. Neil Yorke-Smith,	Delft University of Technology, Chair
	Prof. dr. Jan S. Rellermeyer,	Leibniz University Hannover, Supervisor
	Dr. ir. Sicco Verwer,	Delft University of Technology

Cover: Created using DALL-E 3. Illustration of a battery surrounded by semi-conductors. Retrieved from https://copilot.microsoft.com/.

Abstract

Computers have become an essential part of modern life. They are used in a wide range of applications, from smartphones and laptops to data centers and supercomputers. However, the increasing usage of computers has led to a rise in energy consumption, which has significant environmental and economic consequences.

The hardware of computing systems has become more energy-efficient over the years. However, software remains a largely untapped area for energy optimization. The key to reducing energy consumption in computing systems lies in understanding the impact of software on the overall energy usage of the system. So, the need for tools and methods to measure software energy consumption is crucial. Those tools and methods should be flexible enough to be used in different computing environments.

This thesis reviews the current state-of-the-art tools for monitoring software energy consumption. After a list of their capacities, we identify the more promising tools and methods and compare their performance during experiments using different types of workloads. The thesis also presents a new tool called *Geryon*, which brings together the best techniques found in the tools reviewed. *Geryon* is a flexible and easy-to-use tool that provides multiple estimations at once. Those estimations are computed via simple power models and regression models trained on-the-fly. It also provides a way to plug in an external power model, allowing for more complex power models to be used. The tool is evaluated on the same workloads as the other tools to provide a fair comparison.

Preface

Since I was a child, I have always been fascinated by computers and technology. This fascination led me to pursue my studies in the field of computer science. During my studies, I have also become more aware of the environmental impact of technology and grew interested in the field of sustainable computing. So, when the opportunity arose to work on a thesis on software energy consumption, it motivated me to take on the topic. This topic taught me all the aspects of the energy consumption of a system, the limits that arise when trying to measure a particular software, and how people have tried to overcome those limits.

This thesis would not have been possible without the support and guidance of several people. I would like to express my gratitude to my thesis supervisor, Prof. Dr. Jan S. Rellermeyer, for his invaluable advice, guidance, and support throughout this journey. His expertise and encouragement have been instrumental in the completion of this work. I am also thankful to Dr. Neil Yorke-Smith for bringing the idea of the topic and being the chair of my committee, as well as Dr.ir. Sicco Verwer, thank you for being part of the committee. Finally, I want to thank my family and friends for their unwavering support and encouragement through this journey.

Bertrand Leclercq
Delft, June 2024

Contents

Abstract i

Preface ii

Nomenclature v

1 Introduction 1

1.1 Motivation 1

1.2 Problem Statement 3

1.3 Scope and Approach 4

2 State of the Art 5

2.1 Background 5

2.2 Measuring Power Consumption 6

2.3 Software Tools 8

2.3.1 perf 9

2.3.2 PAPI 9

2.3.3 LIKWID 10

2.3.4 PowerTOP 10

2.3.5 Powerstat 10

2.3.6 Intel Applications 10

2.3.7 Powermetrics 11

2.3.8 GPU SMI 11

2.3.9 CodeCarbon 11

2.3.10 PowerJoular 12

2.3.11 Kepler 12

2.3.12 PowerAPI 12

2.3.13 Scaphandre 13

2.4 Discussions 13

3 Methodologies 15

3.1 Machines 15

3.2 Tools 16

3.2.1 PDU 16
3.2.2 PowerAPI / SmartWatts 16
3.2.3 Scaphandre 18
3.2.4 PowerJoular 19
3.3 Experiments 20
3.3.1 SPECPower test 20
3.3.2 NAS Parallel Benchmarks test 21
3.3.3 Stress test 21
3.3.4 Overhead test 22
3.3.5 Metrics and graphs 23
3.3.6 Data processing 23
3.4 Summary 24

4 Experiments 25
4.1 SPECPower test 25
4.2 NAS Parallel Benchmarks test 30
4.3 Stress test 30
4.4 Overhead test 33
4.5 Discussion 34

5 The Geryon program 36
5.1 The base program 36
5.2 The time ratio 37
5.3 The regression 37
5.4 The external model 38
5.5 The complete program 39
5.6 Summary 40

6 Analysis 41
6.1 Time ratio and regression 41
6.2 External model 43
6.3 Discussion 45

7 Conclusion 47
7.1 Future Work 49

References 50

Nomenclature

Abbreviations

Abbreviation	Definition
RAPL	Running Average Power Limit
NVML	NVIDIA Management Library
API	Application Programming Interface
TDP	Thermal Design Power
HwPC	Hardware Performance Counters

1 Introduction

1.1. Motivation

The year 1945 marked the start of a new era. It was the creation of the Electronic Numerical Integrator and Computer (ENIAC), the first general-purpose computer. This computer, built by John Presper Eckert and John Mauchly, could perform various complex calculations. Since then, computers have evolved significantly. Their evolution could be described into different key characteristics. Computers have become smaller and more portable. They have gone from room-sized machines to pocket-sized devices. Their speed and processing power have also increased exponentially. Compared to the ENIAC, simple smartphones today are much faster and more powerful. They have also become more flexible and have found applications in various fields, from simple calculators to complex simulations. Computers are no longer just for science and engineering. They are used in industries such as finance, healthcare, and entertainment, as well as in research, education, and many other fields. They have become an essential tool in our daily lives and for the development of our society.

To consider the scale of our dependence on computers, we can look at the number of users and devices connected to the internet. In 2023, the number of internet users worldwide reached 5.35 billion [10], which is around two-thirds of the global population and is still growing. For the number of devices connected to the internet, Cisco released a report in 2020 [7] that estimates that the number of devices connected to the internet will reach 29.3 billion by 2023, compared to 18.4 billion in 2018. The evolution to a world where everything and everyone is connected, together with the shift in data storage and processing to cloud data centers housing thousands of servers, further increases those trends.

However, this proliferation comes with a cost. The energy consumption of computers has also increased significantly. For data centers, which have become the beating heart of our digital world, the estimated global electricity consumption in 2022 ranged from 240 to 340 TWh, accounting for approximately 1-1.3% of the global electricity demand [38]. This does not even take into account the energy used for cryptocurrency mining, which was estimated to be around 110 TWh in 2022, accounting for 0.4% of the global electricity demand. In terms of greenhouse gas emissions, data centers and data transmission networks were responsible for around 330 Mt CO2-equivalent in 2020, accounting for 0.9% of energy-related GHG emissions. If we take the IT industry as a whole, it was estimated by a post note from the UK Parliament that its energy consumption was between 4% and 6% of the global electricity demand in 2020 [1].

As all those numbers show, this growing use of computers and its need for energy has a significant impact on the environment. All those devices and data centers have become important contributors. And as our society becomes increasingly digital, this impact will only increase. However, it is not only about the effects on the environment. The energy consumption of computers also has economic repercussions. The expense due to power consumption represents a significant part of the total cost of operating a data center. Finding ways to reduce this impact and make our digital world more sustainable and energy-efficient has become a critical challenge.

On the road to a more sustainable future, the IT industry has started taking action to make energy efficiency their main concern. They have aligned their goals with the Net Zero Emissions by 2050 (NZE) Scenario, which aims to limit global warming to 1.5°C. In this scenario, data center emissions must halve by 2030 [38]. Companies have begun to invest in renewable energy sources to power their data centers. Hardware manufacturers have started designing more energy-efficient components using smaller transistors, improving their architecture and optimizing their power consumption. All the modern processors are now designed with energy efficiency in mind. Alongside the processors, even the other components that are not directly part of the computing capabilities but help the computers to work better, such as cooling systems, are also becoming more efficient. All those efforts have led to a significant reduction in the energy consumption growth rate in data centers during the last decade [24].

However, focusing on the hardware is only one part of the solution. The software running on those devices also plays a significant role in their energy consumption. With the emergence of more complex and energy-hungry applications like artificial intelligence and machine learning, the software's impact on energy consumption has become even more critical. But, most of the software developers are focusing their efforts on improving the performance and functionality of their applications, with little consideration for energy efficiency. Applications are sometimes designed to use more resources to improve performance, which can lead to higher energy consumption.

One of the reasons for this lack of focus is the lack of tools for measuring the energy used by software. Unlike hardware, where energy consumption can be measured directly using tools like power meters, measuring the energy consumption of a single software process is much more complex as those power meters are not able to profile the energy at such a fine granularity. With a per-process energy measurement tool, developers would be able to easily identify the energy-hungry parts of their software and could try to optimize them.

Being able to monitor the consumption of software in real-time and develop more energy-efficient software would be beneficial in many ways. In the context of data centers, the ability to monitor the energy consumption of each application running on the servers would allow data center operators to optimize their energy usage. They could allocate resources more efficiently, schedule energy-intensive tasks during off-peak hours, or even move them to servers powered by renewable energy sources, regroup applications with low energy consumption on the same server and thus reduce the number of servers running, and many other strategies. This ability would not only reduce the energy consumption of data centers but also reduce their operating costs. For end-users, having more energy-efficient software would also be beneficial as it would allow them to use their mobile devices for extended periods without recharging. It could also help them to simply be aware of the impact of their daily usage and their environmental footprint. Even in the world of the Internet of Things (IoT), where devices have limited energy resources, being able to develop more energy-efficient software could help developers bring more interesting functionalities to their devices.

Such software tools would also have a lot of benefits compared to hardware solutions. They would not require any additional hardware, which would make them more accessible and easier to deploy without any need for investment. They would also be more flexible and could be used in a wide range of scenarios, monitoring not only at a system level but also at various levels of granularity, down to the process level. This flexibility would also make it easier to monitor applications that are distributed across multiple devices.

In recent years, more and more of those software-based power meters have been developed. They are based on different techniques and use various metrics to estimate. Some will base their estimation on the thermal design power (TDP). In contrast, others will try to estimate more accurately by retrieving the power consumption of different components directly from the hardware through some internal interfaces. The way they compute their estimations can also vary, with some of them using machine learning models, like regression algorithms, to predict the power consumption of the software based on some features and metrics retrieved from the system. The others will use more traditional computations by implementing some power formulas based on the hardware specifications. Those tools also show differences in terms of their features and capabilities. Some are restricted to a specific operating system or hardware, while others support a wide range of systems. Some will only focus on the CPU metrics, while others will include other essential components like the GPU or the memory. Some can monitor the energy consumption of a single process inside a container or a virtual machine. Some allow users to change the sampling rate or the granularity by regrouping the process of a single application. Even how they present the data can vary, with some just outputting on the command line interface while others offer a way to export the data to a wide range of formats. All those tools have their strengths and weaknesses, and it can be challenging to choose the right one for a specific use case.

1.2. Problem Statement

The main problem is that even if there are many software-based power meters available, none of them seems to be generic enough to be used on a wide range of systems with different hardware configurations. They are often limited to specific hardware or operating systems, or they require a lot of configurations to work correctly. This lack of a generic tool creates a barrier for developers who want to monitor the energy consumption of their software. They will need to spend time searching for the right tool, installing it, configuring it to match their system, and then analyzing the results.

The primary focus of this thesis is the implementation of the most generic software-based power meter that can estimate with the best accuracy the energy consumption of an application running on a computer. It should be generic in a way that it can be used on a wide range of hardware without requiring too many configurations or modifications from the user to be able to work.

In order to reach our goal, we will first need to review and compare the existing software-based power meters. We will need to understand how they work, what are their strengths and weaknesses, including what are their requirement to run, and their limitations in terms of hardware. We will also need to compare their accuracy and their performance regarding the overhead they introduce. This comparison will help us to identify the best techniques and metrics to use to estimate the energy consumption of an application.

Once we have identified the best techniques and metrics, we will be able to design, implement, and evaluate our software-based power meter using the other tools as a base. The thesis will thus focus on the following principal research question:

RQ How to design and implement a generic software-based power meter that can accurately estimate the energy consumption of an application running on a computer?

The research question will be divided into three sub-questions:

RQ1 What are the existing software-based power meters, and how do they work and compare in terms of accuracy, performance, and features?

RQ2 What are the best techniques and metrics to use to estimate the energy consumption of an application?

RQ3 Can we design and implement a software-based power meter that would enhance the existing tools and overcome their limitations?

1.3. Scope and Approach

As the scope of those research questions can be pretty broad, we will need to focus on some aspects and discard others to try and limit our scope. Those aspects include the hardware and the operating systems on which the software-based power meters can run. Regarding hardware, we will restrict ourselves to specific CPUs from Intel and AMD, as they are the most widely used in the market. For the operating systems, we will focus on Linux, as it is the most used in the server market and commonly used in the desktop market by developers. We will also limit ourselves to the software-based power meters that are open-source and freely available, as they are the most accessible and can be easily modified and extended.

In order to answer the research questions, we have structured the thesis into several chapters. In Chapter 2, we will provide the necessary background information about the energy consumption of computers, the power formulas that can be used to estimate the consumption, and the existing power meters, including hardware-based and software-based. This chapter will help us to understand the context of our research and the state of the art.

The following chapters will try to answer the two first sub-questions by conducting experiments and evaluations. In Chapter 3, we will present the methods and configurations used to evaluate the existing software-based power meters. In Chapter 4, we will show the evaluation results and compare the existing tools regarding accuracy and performance. We will also identify the best techniques and metrics to estimate the energy consumption of an application.

Finally, in Chapter 5, we will present the design and implementation of our software-based power meter and how it should enhance the existing tools. It will be followed by Chapter 6, where we will evaluate our tool and compare it with the existing ones. We will also discuss the limitations of our tool and how it could be improved. Finally, in Chapter 7, we will conclude the thesis by summarizing the main findings and contributions and discussing future work.

2 State of the Art

This section presents some background information about the energy consumption of computers and the state of the art of power monitoring tools and techniques.

2.1. Background

To measure the power consumption of a system, we first need to understand what power is and which components consume it. Let's start by defining power and energy. Power is the rate at which work is transferred or converted and is measured in watts (W). In the context of a computer system, it is the rate at which electrical energy is consumed. On the other hand, the energy is the total amount of work done for a given time and is measured in watt-hours (Wh) or joules (J), which equals one watt for one second. So, to calculate the energy consumed by a system, we need to multiply the power consumed by the time it has been consumed. For example, if a system consumes 100 watts for 1 minute, it will have consumed 6000 joules or 1.67 watts-hours.

In a modern computer system, the power is consumed by several components. The first and main power-consuming component is the central processing unit (CPU), also known as the processor. The CPU is the brain of the computer: it executes the instructions, performs calculations, and manages the data. It is the one that processes the code from running software and performs the necessary operations to execute them. The CPU is housed on a motherboard, which is the main circuit board of the computer. The motherboard connects the CPU to the memory, the storage, and the peripherals devices and allows them to communicate with each other. It is also the one that receives the power from the power supply unit (PSU) and distributes it to the different components of the system. Another power-hungry component is the graphics processing unit (GPU), which, as its name suggests, is responsible for processing the graphics of the system. It is used to handle the rendering of images, videos, and animations and is able to do complex calculations. The GPU is usually used in video games, video editing, and machine learning. Besides those components that consume the most power, there is also the memory (RAM) used to store the data that is currently being processed, the storage (hard drive, HDD, or solid-state drive, SSD) used to store the data permanently, and the peripherals devices (input and output devices, network cards, etc.) that are used to interact with the system. Even components that are not directly involved in the computation, like the cooling system (fans, water cooling, etc.), consume power. Moreover, when a system uses more resources, more power will be needed to cool it down.

2.2. Measuring Power Consumption

In a computer system, the power consumption can be measured at different levels, as shown in Figure 2.1. At the highest level, the power consumption can be measured at the system level. This is the total power consumption of the system, including all the components. At a lower level, the power consumption can be measured at the component level. This means that the power consumption of each component can be measured separately. The lowest level is the process level, where the power consumption of each process running on the system can be measured. The processes from the same program can be grouped to measure at the application level. Of course, a process is composed of multiple components, and one component can be used by multiple processes. Measuring power consumption at a higher level is usually easier but does not provide as much information and visibility as measuring it at a lower level. For example, when we measure the power consumption at a system level, we have no information about which component or process is consuming the most power.

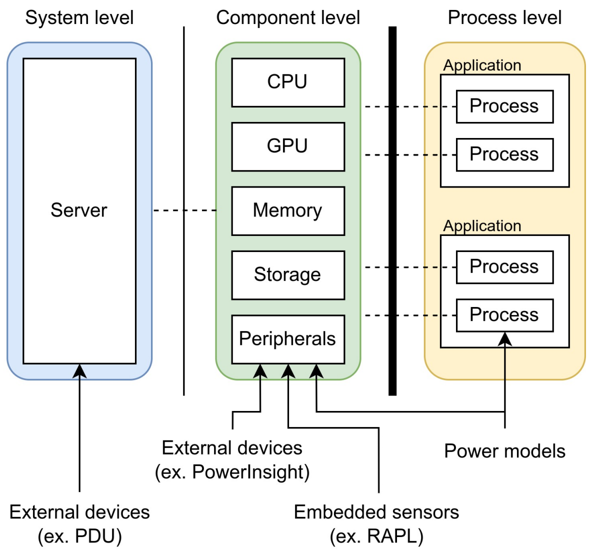


Figure 2.1: The different levels of a computer system where the power consumption can be measured

Nowadays, there are many ways to measure the power consumption of a computer system. All those methods could be classified into three categories: external devices, embedded sensors, and power models.

External devices are devices that are not part of the targeted system and are used to measure the power consumption of the system. The most common external device is the power meter. A power meter is a device plugged between the power outlet and the system to measure the power consumption of the system. It is usually used to measure the power consumption of a single device. Still, some power meters can also be used to measure the power consumption of a group of devices by plugging them into a power strip. Another external device is the power distribution unit (PDU). A PDU is a device that distributes power to multiple devices. It is usually used in data centers to power servers. While its primary purpose is to distribute power, some PDUs provide remote access that we can use to monitor the power consumption of the devices that are connected to them. There are a large number of power meters and PDUs available on the market, each with varying performance and features. One advantage of external devices is that they are usually more accurate than the other methods and have a lower performance overhead. However, they require physical access to the system, and this additional hardware requires an extra cost. Moreover, those devices can only measure the power consumption at a system level and cannot measure the consumption of individual components or processes.

To be able to go one step further and measure the power consumption at a component level, there exist some other external devices that can be used, like PowerInsight [22]. PowerInsight is a device that can be plugged into the power connector of a component to measure its power consumption. It is usually used to measure the power consumption of a single component, like a CPU or a GPU. However, even if they give better granularity than basic power meters, they require more work and knowledge to be installed and used.

This brings us to the second category of power measurement methods: embedded sensors. Instead of adding some additional devices, component manufacturers have embedded sensors directly in their components to measure their power consumption. Those sensors are usually used to monitor but can also be used to control consumption in some cases. The vendors sometimes provide interfaces to access those sensors and retrieve the data.

For example, in 2011, Intel introduced a new hardware feature called the Running Average Power Limit (RAPL). It was created to allow the processor to monitor its power consumption and control it if needed. Those power consumption values are reported as special hardware registers named Hardware Performance Counters (HwPC). They are small registers that count hardware events such as instructions executed, cache misses, etc. They are included in almost all modern processors and are usually used to monitor different aspects of the performance of a program and tune it to be more efficient. [25] With the RAPL, the system is divided into different domains:

- The Core (PP0) domain reports the energy consumption of the CPU cores.
- The Uncore (PP1) domain reports the energy consumption by the uncore components that are close to the CPU (most of the time, the energy consumption of the integrated GPU).
- The DRAM domain reports the energy consumption from the RAM modules.
- The Package (PKG) domain is the sum of the first domains (PP0 and PP1).
- The PSys domain, which is relatively newer (since the Skylake micro-architecture), represents the entire System on Chip (SoC).

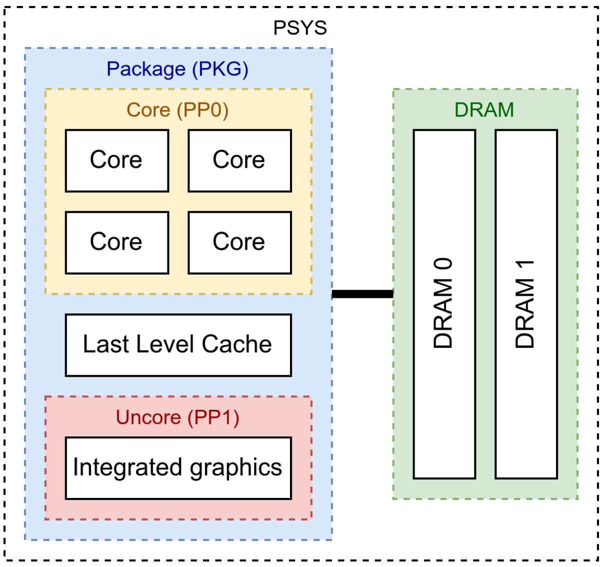


Figure 2.2: The different RAPL domains of a CPU

It seems that there is not that much detailed documentation about how its calculations are done. Also, its implementation has evolved. The first implementation used a model to estimate energy usage based on a set of events. The following implementation, since the Haswell architecture, makes actual power measurement as it is based on fully integrated voltage regulators [13]. A paper has shown that RAPL registers have a high update frequency, approximately every 1 ms, and a low-performance overhead [18]. Another slight advantage of RAPL is that it always runs as soon as the processor is powered on, so there is no need for complex configuration to enable it. In 2017, AMD introduced a version of RAPL with its Zen architecture (17h family) [2]. The implementation is similar to the first model-based version of RAPL.

In the same way, Nvidia has introduced a similar feature for their GPUs. It is an application programming interface called Nvidia Management Library (NVML) that allows users to access GPU device metrics [28]. Those metrics include the current usage, temperature, and power consumption. It also provides data from connected devices, like memory or fans. The power consumption data is available on devices since the Fermi family (2010). Like RAPL, there is not much information on how the power consumption is retrieved or calculated, but there are some papers that show the existence of a power sensor on the GPU. [4, 32]

In case the components do not have embedded sensors and the use of external devices is not possible because of the cost or the physical access, there is a third category of power measurement methods that can be used: power models. Power models are mathematical models that estimate the power consumption of a system based on some metrics. Those metrics are generally the usage of the components (CPU, GPU, etc.), but they can also be the temperature, the frequency, etc. Some of those models are just simple linear models that assume that the power consumption is proportional to the usage of the components. Others are more complex and may even use machine learning algorithms to estimate power consumption. Compared to the other methods, power models are usually less accurate, but they are easier to use. However, they are not always generic and may require calibration to be accurate on a specific system.

Power models sometimes rely on information provided by the component manufacturers to make their approximations. For example, vendors can provide a power consumption value for a specific component under a constant workload. This value is called the Thermal Design Power (TDP) and is the maximum amount of heat the component can dissipate. Even if it is not a direct power consumption value and can sometimes be exceeded, it still helps to get a good estimation when the component is at full load. For example, the power usage of a CPU can be estimated by the TDP value multiplied by the CPU usage. By adding the execution time of the program to the calculation, we can get the total energy consumption.

Even if the three categories of power measurement methods are usually used to measure the power consumption at a system level or a component level, it is also possible to go one step further in terms of granularity and even measure the power consumption of a specific process. This could be done in two ways: usage-based or event-based. The usage-based method is the simplest one. It consists of measuring the overall power consumption of the system and the CPU usage of the processes running on the system. By multiplying the power consumption by the CPU usage, we can estimate what part of the power consumption is due to a specific process. The accuracy of this method depends on the accuracy of the power consumption measurement, but also on the way the CPU usage is calculated by the system. The event-based method is more complex. It consists of monitoring the power consumption of the system and the hardware events of the processes running on the system. By correlating the power consumption with the hardware events, we can create a regression model that estimates the power consumption of a process based on its hardware events.

All those methods have their advantages and disadvantages. The external devices are more accurate but require physical access to the system and add cost. The embedded sensors are easier to use but may not be available on all components. The power models are easy to use but are less accurate and may require calibration to work on a specific system.

2.3. Software Tools

After understanding what power is and the different ways to measure the power consumption of a system, we can now present some software tools that can be used to monitor the power consumption of a system. All of those tools rely on the methods presented in the previous section. This list is not exhaustive, but it contains some of the most popular tools we have come across that are, for most of them at least, still maintained, well-documented, and open-sourced.

Before presenting the tools, it is essential to note that some tools require special permissions to access the power consumption data. This requirement is because the power consumption data is considered sensitive information and is usually only accessible by the system administrator. For example, on Linux, access to performance and observability operations is limited to users with specific capabilities (CAP_SYS_ADMIN). This limitation can be changed to allow users without the capability by modifying the system settings named `perf_event_paranoid`. Another way to access the power consumption data is to use the `powercap` interface from the Linux kernel. This interface is a set of files in the `/sys/class/powercap` where the power consumption data can be read. To access those files, the user needs to have the correct permissions. Since version 5.4.0 of the Linux kernel, the powercap attributes have been made readable only by the root user. Thus, programs that rely on the powercap interface must be run as root.

2.3.1. perf

The Linux perf tool [15] is a performance analyzing tool capable of lightweight monitoring and profiling the performance of the system or a program. It was introduced in the Linux kernel version 2.6.31, released in 2009, and is part of the kernel, which makes it available on all Linux systems. It can handle most HwPC registers available on the system, as well as tracepoints and dynamic probes. Tracepoints are markers that are placed at specific points in the code to trace the execution of the program. The command will then collect the data from the tracepoints, including helpful information like the timestamp or the slack traces. Dynamic probes allow users to create tracepoints on a running program without modifying the source code for the kernel (kprobes) or user-space (uprobes) dynamic tracing. Five years after its introduction, in 2014, the perf tool was extended to support the RAPL registers with version 3.14 of the Linux kernel. With this new support, the perf tool is able to monitor the power consumption of the system or a program. It is composed of multiple subcommands, each one having its purpose.

The most valuable subcommands for power monitoring are `stat`, which retrieves and directly shows a summary of the performance counter statistics, and `record`, which records the performance counter statistics for later analysis. The output of the `stat` subcommand for the RAPL registers will return the energy consumption in joules. The subcommands allow users to specify a lot of options, like the period of the measurement, the interval between each sample (otherwise, it will only print the summary at the end of the execution), or the events and processes to monitor. However, even if the perf tool has the option to specify a process to monitor, as the RAPL registers values are energy consumed for each of its domains, it will only return the power consumption at a domain level during the execution of the process. If it is mixed with other hardware events that can be monitored at a process level, like instructions executed, the power consumption will be blank in the output.

2.3.2. PAPI

Performance Application Programming Interface (PAPI) [39] is not a tool like the others presented in this section. It is a toolkit that allows access to hardware performance counter information through a consistent interface and methodology. It was introduced in a paper released in 2001 by the Innovative Computing Laboratory at the University of Tennessee.

Compared to the other tools, its goal is not to estimate the power consumption or monitor the performance of the system but to provide an easy and high-level way to access the hardware performance counters for other applications. Each modern processor has a set of various hardware performance counters that can be different from one processor to another, even from the same manufacturer. PAPI will summarize the common counters into a set of standard events so that the developer does not need to know the specific counters of each processor. It also takes care of possible counter overflow and multiplexing.

Thanks to the liaison with industry companies, it can support new architecture near their release.

2.3.3. LIKWID

The LIKWID Performance Tools [36] is a set of command line tools designed for Linux systems that can analyze and optimize the performance of the system across various aspects. The paper on this toolset was published in 2010, and it has been maintained since then.

Among these tools is one called `likwid-powermeter`, which accesses the RAPL registers and can deliver the energy consumption of all the RAPL domains for a certain period. Similar to perf, if an application is specified, the output would be the energy consumption of all domains for the time of execution of the application. The output of the tool contains the energy consumed in joules and the power consumed in watts. The tools also provide options to output the temperatures of all the processor cores.

Another tool from the LIKWID set called `likwid-perfctr` can measure any HwPC events available on the system, including the RAPL registers, for a given process. This last tool can work in different modes: it can measure the events from an application without any code modification needed, measure the events from a specific function or part of the code using a marker API, measure the events for a specific duration independently of any application, or measure the events with a particular frequency.

2.3.4. PowerTOP

PowerTOP [37] is a Linux tool that measures and analyses the power consumption and management of a system. It was released by Intel in 2007. At the time of writing this report, the last release was in 2022, but there is still some activity on the GitHub repository. In addition to monitoring the power consumption, it also looks at the drivers and the kernel options. It helps to find the programs or the settings that are responsible for excessive power consumption and provides recommendations to reduce it.

The tool has an interactive mode that shows various statistics about the processes, the cores, and the devices of the system. This mode also provides easy access to power management settings. Otherwise, it can be used to generate a report in HTML or CSV format. Both modes will display an estimation of the power consumption of the process in watts.

2.3.5. Powerstat

Powerstat [20] is another command line tool for Linux based on RAPL registers or battery statistics. The first version was released in 2011, and it is still maintained. Compared to the other tools, it is simpler, focuses on the power consumption of the whole system, and does not care about the other HwPCs. Given several samples and a sampling interval, it will report several statistics for each sample, including the average power consumption (in watts). Finally, the tool also reports statistics like the average and the standard deviation of the power consumption for the whole run.

2.3.6. Intel Applications

The Intel Power Gadget [35] program is a graphical application for Windows and macOS that can be used to monitor the power consumption of an Intel system. When it is open, it shows a simple vertical interface made of different plots: the power consumption of the different RAPL domains in watts, the frequency of the CPU, the temperature, and the utilization of the CPU. The application provides a way to log a measurement to a CSV file. It also comes with a command line tool, Intel PowerLog, that can also generate a similar CSV file and can be used in other scripts.

At the end of 2023, the Intel Power Gadget application was discontinued in favor of the Intel Performance Counter Monitor (Intel PCM) [16], which was first released in 2017. Intel PCM is composed of two parts: an application programming interface and a set of tools that uses this API to monitor the performance and energy consumption of Intel processors. Among those tools, there is a command line tool called `pcm-power` that can monitor various energy-related metrics of the processor, like the energy states of the processor and the memory or their energy consumption. There is also the central tool called `pcm` that monitors various metrics of the processor, including the energy-related ones.

One advantage of this new tool compared to the old application is that it is available on all the major operating systems (Windows, Linux, and MacOS). While this new tool does not have a built-in graphical interface, it provides ways to be connected to third-party frond-ends like Grafana to visualize the data.

2.3.7. Powermetrics

The powermetrics tool [3] is a command line tool bundled with MacOS, the same way perf is with Linux. It is used to retrieve metrics related to the CPU, like temperature, utilization, or power consumption of the CPU and other connected components, including integrated GPU and memory. There is not much information about the tool from Apple, as they only direct users to the Manual page of the command available on the operating system. The only information we found was from other tools that use the powermetrics tool to retrieve the power consumption on Apple devices.

For the devices that are still using Intel processors, the tool will return the power consumption values of the RAPL domains available. For the devices that are using the new Apple Silicon chips, the tools will return a set of values organized in a similar way to the RAPL domains. First, there is a power value for each of the CPU clusters, one for the efficient cores and one for the performance cores. Then, there are the values for the integrated devices, such as the GPU and the memory (DRAM). The last value returned, called the package power, is the total power consumption of the system, which includes the previous values and the power consumption of other components.

2.3.8. GPU SMI

The Nvidia System Management Interface (nvidia-smi) [29] is a command line tool used to monitor and manage GPU devices manufactured by NVIDIA. It is based on the Nvidia Management Library. It provides basic information about the GPU, like the temperature, the memory usage, and the utilization of the GPU. It can also retrieve the metrics about the power consumption of the GPU. It has some options to specify a frequency for the output, to change the output format, or to select which properties it needs to retrieve. The best use of this tool is to monitor the usage of processes that have extensive uses of GPU. Those processes come from various types of applications like video games, video editors, or scripts to train deep learning models. As it only focuses on the GPU, it can work well in conjunction with other tools that monitor the CPU or other components.

Of course, this tool is only available on systems with an Nvidia GPU. However, other manufacturers have similar tools, like AMD with the ROCm suite. Inside this suite, there is a tool called `amd-smi` that has the same purpose as the Nvidia tool but for AMD GPUs. It can also retrieve the power consumption of the GPU and other metrics like the temperature or the utilization.

2.3.9. CodeCarbon

CodeCarbon [9] is a Python library released in 2020 by volunteers from MILA, the Montreal Institute for Learning Algorithms. It is a tool that estimates the amount of CO2 that is emitted during the execution of the code. The motivation behind this tool is to raise awareness about the environmental impact of the code, mainly for Machine Learning models, which are being used more and more in various domains. The tracker can be used in two ways: either through the command line interface to keep the code unchanged or by using any of the provided Python integrations, such as the tracker object, the decorator, or the context manager.

Its calculation can be divided into two parts. First, it will estimate the power consumption. For the estimation, CodeCarbon uses multiple sources. It retrieves the power consumption of Intel CPUs from the RAPL registers via the Intel Power Gadget (on Windows and MacOS) or the powercap interface (on Linux). For Apple Silicon Chips, the tool will use the powermetrics tools. It also uses an NVML Python library to retrieve the metrics of the Nvidia GPUs. For the rest, or when the tools are not available, it will fall back to a simple calculation based on the TDP or simply use a default value like the ratio of 3 Watts for 8 GB it uses to estimate the power consumption of the RAM.

Then, when the power consumption is estimated, it will look at the energy mix of the country where the code is executed. Using this data and the carbon intensity for each energy source, it will find the amount of CO2 that is emitted for a kilowatt-hour. By multiplying the result with the power consumption it has estimated earlier, it will get the amount of CO2 that is emitted during the execution of the code.

2.3.10. PowerJoular

PowerJoular [27] is a simple Linux software, written in Ada, that allows monitoring the power consumption of different processes. It was presented in a paper in 2022. On top of processors that support RAPL, it also supports Raspberry Pi devices and Asus Tinker Board through pre-trained regression models to estimate the power consumption of these devices. To estimate RAPL-based devices, it will do something similar to Scaphandre by using the CPU time of each process with the overall CPU time and computing the ratio for each process.

There is also another program called JoularJX, which is written by the same author. This Java-based agent monitors the power consumption at the source code level. It does something quite similar to the CodeCarbon library to estimate the power consumption: in addition to Linux, this software can be used on Windows software, where it collects the metrics with the Intel Power Gadget tool, or on MacOS, using the bundled `powermetrics` tool.

2.3.11. Kepler

The Kubernetes-Based Efficient Power level Exporter (Kepler) [8] is a tool released in 2022 that monitors various statistics from an application running on a Kubernetes cluster. One interesting feature of Kepler is that it can estimate the power consumption of a process that is running inside a container where containers are sharing the same resources. To make its estimation, it first uses eBPF to probe hardware performance counters and other process statistics. It will also extract the power-related metrics for the host components. It can also use Machine Learning models to estimate energy consumption based on the system metrics when no counters are available. When it has all the data it needs about the resource utilization and the power consumption, it will calculate the power consumption of the process using the utilization ratio of the process and the power consumption of the system. Its last step is to aggregate the power consumption of all the processes running in the same container to get the total power consumption for each container.

2.3.12. PowerAPI

PowerAPI [33] is a Python toolkit to help create power meter programs to estimate software consumption. It was presented in a paper in 2013. It supports a wide range of sensors to retrieve data and export power consumption in many ways. For PowerAPI, a power meter comprises two parts: a sensor and a formula. Both parts can be executed on different machines. There are various ways to export the data from the sensor to the formula and then to the user.

The sensor is responsible for collecting the metrics that are correlated to the power consumption. The provided sensor is the HWPC Sensor, written in C, which accesses the HwPC registers through the perf API to collect data on CPU events for the monitored software and the RAPL metrics. To monitor a specific software, the sensor relies on Control Groups (CGroup), a Linux kernel feature allowing users to group processes and manage their resources.

The formula is responsible for computing an estimation of the power consumption from the raw data collected by the sensor. The formula can be executed on a machine different from the sensor. It can process data in real-time or after the sensor has been executed. The current formula, presented in a paper in 2020 and called SmartWatts [12], will use multiple regression models, selected based on the CPU usage, to estimate the power consumption of the monitored software using several metrics collected by the sensor as input. At each time interval, the sum of the counters is collected together with the power consumption of the system. Those values are used as the input and the target of the models. Of course, as there are multiple models based on CPU usage, the data are also split in the same way. When a model is trained, it is used on the individual counters of each monitored CGroup to estimate their power consumption. If the estimation becomes too inaccurate, the model will be retrained on the latest data.

2.3.13. Scaphandre

Scaphandre [31] is a Rust software release in 2020 that monitors performance, including power consumption, at different levels. It contains two different sensors to retrieve the RAPL metrics: one that works on Linux via the powercap interface and one that works on Windows via an RAPL MSR-based driver. It also provides different ways to export power consumption data in different formats, like Prometheus, JSON files, or just printed in the terminal.

When the tool is started, it will create the required exporter and sensor. It is the exporter that will ask the sensor to retrieve the data with a specific frequency. The sensor will read the various metrics it needs to compute the power consumption of a process. First, it will use the powercap interface to retrieve the power consumption of the system from the RAPL registers. Then, it will retrieve the usage statistics of the CPU and of all the processes that are running on the system. It will then compute the power consumption of each process by using the RAPL metrics and the CPU usage ratio of each process. Some options are available to the user to specify the frequency of the measurements or to filter the processes that need to be monitored. If the user asks for a specific application, it will find all the processes that are related to the application and aggregate their power consumption. When exporting the estimations, the tool will also provide the power consumption it retrieved for the system per domain and, for some exporters, other metrics from different levels.

2.4. Discussions

The power consumption of a computer system is an essential metric to monitor and is caused by the different components of the system. As we have seen, there are many ways to measure the power consumption of a system with different levels of granularity. All those methods and tools have their advantages and disadvantages.

External devices are the most accurate way to measure the power consumption of a system, but they require additional hardware and physical access. So, embedded sensors, when they are available, combined with power models, are the easiest way to measure the power consumption of a system, even if they can be less accurate. They also do not require physical access to the system, which allows them to be used on remote systems.

While this list gives us a good overview of the existing software-based power meters, to fully answer our first research question (RQ1), we need to compare them and test their accuracy and performance. However, we still need to keep in mind that part of our main research question is the capacity to monitor the power consumption of a specific software.

Tools like perf, PAPI, LIKWID, Powerstat, and Powermetrics are helpful to get access to the RAPL values and other HwPCs in an easy and consistent way. Their main goal is to help developers, or even users, gain access to those values without knowing the specific identifiers or methods that can differ from one processor to another and from one operating system to another. This is why some of them, like perf, are used by the other tools in the list that need to access the RAPL registers to make their estimations. If we refer to the different levels of power consumption monitoring, shown in Figure 2.1, those tools are at the component level. They can provide the power consumption of the CPU, the GPU, the memory, etc., but they can not estimate the power consumption of a specific process. Moreover, they have limited support for operating systems, with the first three tools only available on Linux and the last one on MacOS. Similar things can also be said about Intel Applications and the GPU SMI tools. While they are available on multiple operating systems, they still only provide component-level metrics, with also the limitation that they are focusing on hardware from a specific manufacturer, as they were created by them.

CodeCarbon and JoularJX are libraries for different programming languages that can estimate the consumption of a part of the code. This means they can measure the power consumption at a lower level than the process level. However, they are not generic; they only work for their specific language. PowerTOP and Kepler focus on particular features: the first is more about power management and finding issues that lead to excessive power consumption of the system, and the second is more about monitoring the power consumption of a containerized application.

This leaves us with three tools we will focus on during our comparison: PowerAPI and its SmartWatts formula, Scaphandre, and PowerJoular. All three tools are open-sourced and still maintained. They are limited in their operating system support, as two of them are available on Linux only, and Scaphandre is the only one that can work on both Linux and Windows. However, All three of them can estimate the power consumption of a specific process or an application, which is what we are looking for. As we said in the introduction, we will only focus on Linux, so their operating system limitation is not an issue for us, but it is still something to be aware of.

We still need a precise baseline to compare them and test their accuracy. For that requirement, we will use a PDU to measure the power consumption of the system and compare it with the estimations of the tools.

3 Methodologies

In the previous chapter, we reviewed the different methods and tools that can be used to monitor the power consumption at various levels of granularity on a machine. We ended the chapter by choosing three tools in line with our main research question, as they can monitor power consumption at the process level. In this chapter, we will present an in-depth explanation of how we set up the tools and the experiments we ran to compare them in order to answer our first two research questions.

3.1. Machines

Before going into the details of the tools and the experiments, it is essential to present the hardware configuration of the machines we used. For the experiments, we used three different machines: one with a single Intel processor, one with a single AMD processor, and one with two AMD processors. All those machines run in the same cluster and are connected to the same PDU. To distinct them, we will refer to them by their hostname in the cluster: Headnode, Asus, and Node03. The Headnode machine uses an entry-level Intel Xeon Bronze 3204 with a low TDP and a low number of cores, making it an excellent candidate to get some interesting insight. The Asus machine is the most powerful of the three, with two AMD EPYC 9354 processors, each with 64 logical cores. Finally, the Node03 machine is in the middle, with its AMD EPYC 7313P processor.

A list of the machines with their specifications is available in the table 3.1.

Name	Headnode	Asus	Node03
CPU	Intel Xeon Bronze 3204	AMD EPYC 9354	AMD EPYC 7313P
Serie	Cascade Lake	Fam 19h (Zen 4)	Fam 19h (Zen 3)
# cores (logical)	6 (6)	32 (128)	16 (32)
Clock (min / base / max)	800 / 1900 / 1900 MHz	0 / 3250 / 3800 MHz	0 / 3000 / 3700
TDP	85 W	280 W	155W
Release year	2019	2022	2021
OS	Ubuntu 22.04		
Kernel	Linux 5.15.0		
RAPL	PKG[DRAM]	PKG[Core]	PKG[Core]

Table 3.1: Machines used for the experiments

3.2. Tools

Now that we have presented the machines, we can go into the details of the tools we used and how we set them up. Some shared manipulations are needed for all these tools. First, the RAPL and MSR kernel modules must be loaded, which can be done using the `modprobe` command. Then, as we explained in the previous chapter, to access the power consumption and other information, the tools need to read some files in `sysfs` and `procfs`. Some of those files are restricted, so the user launching the programs must have the correct permissions to access them. To make our tests easier, we directly ran the programs as root.

3.2.1. PDU

As we needed a reference to compare the tools, we used a Power Distribution Unit (PDU) to measure the power consumption of the machines. In our case, as all the machines are from the same cluster, they are all connected to the same PDU. To retrieve the metrics from the PDU, we need to use the Simple Network Management Protocol (SNMP) [14]. The `snmpget` [23] command is the easiest way to obtain such information. We only need to specify the object identifier (OID) of the metric we want to retrieve.

Of course, for our experiment, the values needed to be collected at a specific interval. So, we created a small shell script that will run the `snmpget` command in order to retrieve the power consumption of the machines every second with a variable time limit to end the script. The obtained value is then stored in a text file with the timestamp.

As we also needed the power consumption of the monitored application running on the machines to make the comparison and evaluation, we had to make some assumptions. We assumed that when an experiment was running, the monitored application was the only software that would run on the machine. This assumption is not entirely true, as some background processes are still running. Still, we considered that the power consumption of those processes was negligible compared to the power consumption of the monitored application. With this assumption, we were able to compute the power consumption of the software by taking the difference between the value and the minimum value from the run, as we assume that the minimum value is the idle power consumption of the machine when the software is not running.

3.2.2. PowerAPI / SmartWatts

The first of the three tools we compared was PowerAPI. The PowerAPI framework is composed of two processes: a sensor and a formula. The sensor relies on Control Groups (CGroup) to get the counters for one specific application. It is a Linux kernel feature that allows one to organize the processes into groups and be able to monitor or limit the resource usage of the group. So, to start, the program uses its sensor to retrieve HwPC counters that are correlated with the power consumption for the monitored CGroup, together with the RAPL values of each socket. Then, at each time step, the formula will compute the sum of the HwPC counters for all the CGroup and store the sum with the RAPL value in a history buffer. For each socket, if there is enough data in the history buffer, it uses them as input to train an Elastic Net regression model [40] from the Scikit-learn library [30] with the RAPL value as the target. The Elastic Net is a regression model that combines the L1 and L2 regularization of the Lasso and Ridge regression. For better performance, instead of just training and using only one model, the program creates one hundred models, each one with its history buffer, and at each step, it will choose the correct model and buffer based on the current CPU frequencies. If a trained model is available during a step, it will be used to predict the power consumption of each monitored CGroup using their HwPC counters. The intercept share of the process is then computed and added to the prediction. Finally, at the end of each timestep, the model is used to calculate a prediction for the complete socket using the aggregated sum of the values. Then, the prediction is compared with the current RAPL value to determine whether the model is accurate enough. If the error is above a threshold, the model is retrained on the current history buffer, which only includes the most recent data. An overview of the structure of the PowerAPI program is available in the figure 3.1.

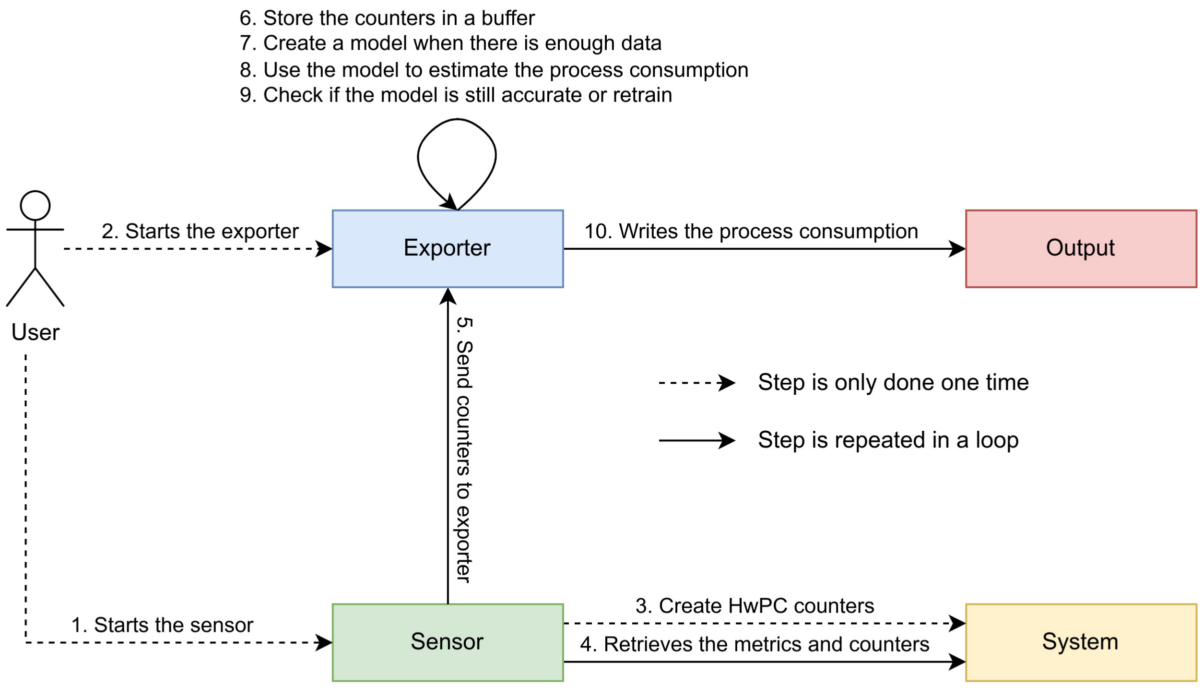

6. Store the counters in a buffer
7. Create a model when there is enough data
8. Use the model to estimate the process consumption
9. Check if the model is still accurate or retrain
User
2. Starts the exporter
Exporter
10. Writes the process consumption
Output
5. Send counters to exporter
Step is only done one time
Step is repeated in a loop
1. Starts the sensor
Sensor
3. Create HwPC counters
4. Retrieves the metrics and counters
System

Figure 3.1: Structure of the PowerAPI program

The two processes can be separated, and usually, only the sensor process is put on the machine to reduce the overhead caused by all the programs running together. To transfer the data from the sensor to the formula, we first used SSH port forwarding to connect the sensor to a MongoDB database running on a local machine, with the formula running in streaming mode as the data from the sensor was coming. However, as the connection is not always stable, we decided to completely separate the two programs by setting the output for the sensor to a CSV file. When the experiment is finished, we transfer the CSV file to the local machine to run the formula in deferred mode. Only during the experiments on the overhead of the tools did we run the formula on the same machine as the sensor, as we needed to compute the overhead of the complete power consumption computation.

Nowadays, Linux uses CGroup v2, but the current version of PowerAPI only supports CGroup v1. Fortunately, Linux provides a simple way to enable CGroup v1 while also keeping CGroup v2 by adding kernel parameters to the boot configuration. This manipulation has been done on all the machines we use. Then, we created an arbitrary CGroup, and when we started the application, we used the `cgexec` command, which will run the provided command and have its process to the CGroup together with any child processes it creates. The sensor will then be able to get the counters for the application. As the sensor is monitoring the CGroup as a whole, if multiple applications running simultaneously need to be monitored separately, we need to create a CGroup for each software.

Regarding which event we want the sensor to retrieve, we used the default configuration provided on the PowerAPI website. The only option we changed was the interval of the measure, which we set at 2 seconds unless specified otherwise. So following the default configuration, for the system events, there are two groups: one for the RAPL, which looks at the RAPL_ENERGY_PKG event on only one core per socket (as the value is the same for any core), and one for the Model-Specific Register (MSR) events (TSC, APERF, and MPERF). For the monitored groups, it depends on whether the processor was an Intel or an AMD. For the Intel, the events were:

- CPU_CLK_UNHALTED:REF_P
- CPU_CLK_UNHALTED:THREAD_P
- LLC_MISSES
- INSTRUCTIONS_RETIRED

And for the AMD, the events were:

- CYCLES_NOT_IN_HALT
- RETIRED_INSTRUCTIONS
- RETIRED_UOPS

To find the events encoding from the events name, the sensor uses the ‘lifpfm4’ library internally. Inside the library, each family of processors has a file containing the events encoding. The issue with that way is that even the latest version of the library does not include the complete encoding for all recent processors. For example, the library does not recognize the AMD Family 19h (Zen 4) processor like the one used on the Asus machine for the RAPL event encodings. To fix this, we used a modified version of the library where we added the case of the missing family. We then rebuild the sensor using the modified library.

For the formula, some configurations were needed to calibrate the software to the machine. Those configurations are the basic information of the CPU: the base clock, the base frequency, and the Thermal Design Power (TDP). The other configurations are the minimum number of samples before a regression model is trained and the maximum number of samples to keep in memory if a model needs to be retrained. For both configurations, we used the default values of 10 and 60, respectively. This means that the formula will start to train a model after 10 samples and keep the last 60 samples in memory. The last configuration is the error threshold, the maximum error allowed before the model is retrained. We set this value to 5 watts, a reasonable value for our experiments.

3.2.3. Scaphandre

The second tool, Scaphandre, is a much simpler program to set up as it requires less configuration. The program is divided into two parts: the sensors and the exporters. Each part has different implementations, and the user can choose which one to use. The program has a `mod.rs` file for each part containing the common logic that any sensor or exporter needs to implement so that communication between the two parts works correctly.

The sensors are the part of the program that retrieves the metrics and information of the systems. When it is initiated, the sensor will generate a topology of the system, including the CPU sockets and cores. It will also initiate a `ProcessTracker` instance that is used to keep track of the processes running on the system and their information. The difference between the sensors is the way to find and retrieve the RAPL counters, which can vary depending on the operating system. We will only focus on one, the `powercap_rapl` sensor, which retrieves the RAPL counters on a Linux system by reading them from the `sysfs` file system. The structure of the Scaphandre program is available in the figure 5.1.

The exporters use the information retrieved by the sensors and write them to the respective output. When we start the program, we must specify which exporter we want to use. The exporter will then initiate a sensor based on the operating system, and then it will loop. During the loop, it will ask the sensor to refresh the metrics and then read them. The data that is written to the output depends on the exporter. We will focus on two of them, the `stdout` and the `JSON` exporter. The `stdout` exporter writes a summary of the metrics of the top consumers of the system to the standard output. In contrast, the `JSON` exporter will write the metrics of all the processes together with metrics from the host and the CPU sockets to a JSON file. The reason we will focus on those two exporters is that the `stdout` exporter will provide us with a quick way to see the metrics of the system, while the `JSON` exporter will be used to save the metrics during our experiments.

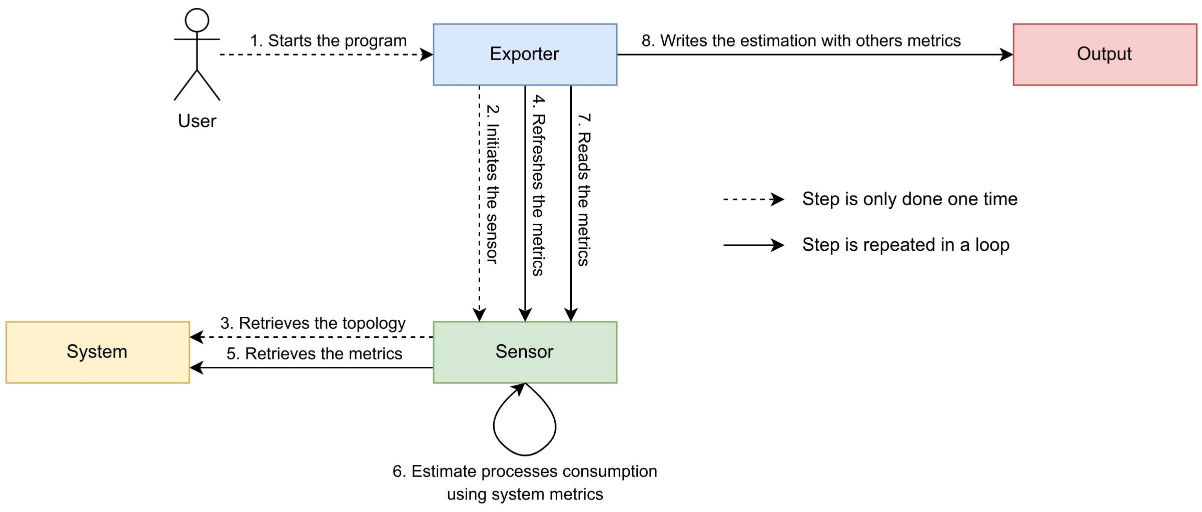

User
1. Starts the program
Exporter
8. Writes the estimation with others metrics
Output
2. Initiates the sensor
4. Refreshes the metrics
7. Reads the metrics
Step is only done one time
Step is repeated in a loop
System
3. Retrieves the topology
5. Retrieves the metrics
Sensor
6. Estimate processes consumption using system metrics

Figure 3.2: Structure of the Scaphandre program

To monitor a specific application, both exporters allow the user to provide a regex to match the name or the execution path of the application and filter the processes shown in the output instead of using the top consumers or all the processes. We can provide a regex. With that regex, it will filter the processes on the system to keep only the ones related to our software. The only configuration we need is to choose which exporter to use to output the data. There are several exporters available, like Prometheus or Riemann, but we used the JSON exporter as it was the most straightforward to use. The program computes and stores a lot of data we do not need, like the power consumption of the whole system or each socket, the resource usage of each process, etc. As we only needed the power consumption of the software we were monitoring, we modified the code to remove the data we did not need so that the file would be smaller and easier to read for small experiments. When saving the data, the program writes JSON dictionaries for each step without any separator, so we made a small script to add those separators to make the JSON file a valid file that our scripts can parse. Like PowerAPI, we also set the measure interval to 2 seconds unless specified otherwise.

3.2.4. PowerJoular

The last tool, PowerJoular, is the simplest program to use compared to the others as it does not have any special configuration to set up. To make its estimation, it will simply compute the ratio between the time the software is running and the time the CPU is busy. To do so, it uses the following equations:

$$process\_time_i = (utime_i + stime_i) - (utime_{i-1} + stime_{i-1}) \tag{3.1}$$

$$cpu\_utilization_i = \frac{cbusy_i - cbusy_{i-1}}{ctotal_i - ctotal_{i-1}} \tag{3.2}$$

$$process\_utilization_i = \frac{process\_time_i}{ctotal_i - ctotal_{i-1}} \tag{3.3}$$

$$process\_power_i = \frac{process\_utilization_i \times total\_power_i}{cpu\_utilization_i} \tag{3.4}$$

Where i is the current time step. First, we need to calculate the time of a process using the `utime` and `stime` values of the process, which are the time spent by the process in user mode and kernel mode, respectively (Equation 3.1). Then, we will retrieve the total time the CPU was running (`ctotal`) and the time the CPU was busy (`cbusy`). With all those values, we first compute the total

CPU utilization (Equation 3.2) and the utilization of the process (Equation 3.3). Then, we use the ratio of the two results to multiply with the total power, computed using the RAPL values, to calculate the power consumption of the process (Equation 3.4). The program will do this calculation at each time step and output the result.

The only configuration it needs is the name of the software to monitor and the output file, which has to be a CSV file as it is the only format it supports. Based on the filename provided, it will create multiple files: one for the overall power consumption and one with a modified name for the power consumption of the program. To collect the PID of the software and monitor them, the program relies on the `pidof` command internally and stops when the software is not running anymore. As some of our experiments would start and stop various processes, we modified the code to continue to look for the PID of the software even if it is not running. We also changed the internal command to `pgrep` to use a regex to match each possible name of the software we want to monitor. However, as there is no way to change the measuring interval, we let the default value be 1 second.

3.3. Experiments

Now that we have explained the configurations and modifications we made to the tools, we can present the various experiments we ran to compare the tools and evaluate their performances. The experiments were designed to help us answer our first two research questions. We have four different experiments: the SpecPower test, the NAS Parallel Benchmarks test, the stress test, and the overhead test. The results of the experiments will be presented in the next chapter.

3.3.1. SPECPower test

The first thing we wanted to do was to compare the accuracy of the tools. To do so, we decided to use a benchmark known for the evaluation of the power consumption of a machine: the SPECpower_ssj2008 benchmark [21]. This is described as the first industry-standard benchmarks that measure and evaluate the power consumption and performance of server-class computers. The benchmark consists of five different Java programs that need to run simultaneously.

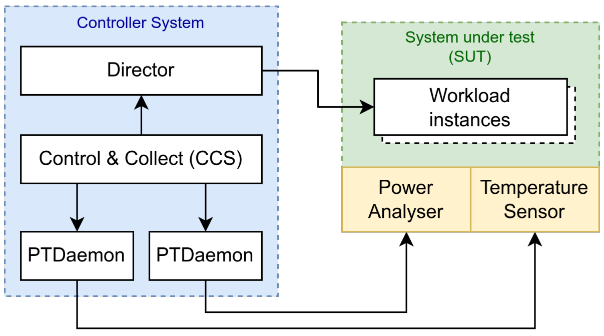


Figure 3.3: Structure of the SPECpower_ssj2008 benchmark

The first two programs are generally used to connect to the power and temperature sensors. As we only care about the workload of the benchmark, but those two programs are needed to run the benchmark, we configured those programs to use dummy sensors.

The third program is the workload simulator. This program simulates many operations at different intensities, starting from 100% of the workload and decreasing to 0% by steps of 10%. The program runs 14 workloads in total, including three warm-up workloads at 100%. Each phase lasts about 5 minutes, which leads to a total duration for the complete run of around 1 hour and 15 minutes. This third program is controlled by the fourth program, the director, who starts the workload program and controls the different phases.

The last program is the control and collect program. As the name suggests, the program connects and retrieves the data from the two sensors and communicates with the director to start the workload program when everything is correctly set up. In a typical setup with actual sensors, only the workload program is run on the monitored server, and the other programs are on a different machine. However, as we used dummy sensors and our goal was mainly to have the workload running, we ran all the programs on the same machine.

We created a small shell script to start all the programs together. We first launched the power monitoring tools we wanted to compare with the correct configuration to point them to the workload process. Then, we started the different parts of the benchmark and ensured the workload program was correctly put in the CGroup. The script will kill all the tools after a slight delay when the benchmark is finished.

3.3.2. NAS Parallel Benchmarks test

Another experiment we wanted to run was the NAS Parallel Benchmarks (NPB) [5]. Besides being a well-known benchmark suite, it is already used in several energy-related experimental studies [17, 12]. The first study was a comparison of different software-based power meters with a focus on CPU and GPU, and the second study was the one presenting SmartWatts. Thus, including this benchmark to compare our results with those papers seems logical.

This suite of benchmarks is a small set of programs that were designed to help evaluate the performance of parallel supercomputers. Each of the benchmarks is divided into multiple classes, representing the complexity of the problem. For example, in some benchmarks, the size of the input given to the program is increased based on the chosen class. The classes generally range from A to F, with A being the smallest or least complex and F being the largest or most complex. The gap of complexity between each class is approximately 4x the previous from A to C, and 16x from D to F. (Two other classes also exist: one that is quick, small, and recommended for test purposes only and one that was used for old workstations and is now considered too small).

To choose which benchmarks we wanted to run, we took the ones that were used the most in the two papers. This results in the following set:

- EP: Embarrassingly Parallel,
- MG: Multi-Grid,
- LU: Lower-Upper Symmetric Gauss-Seidel,
- IS: Integer Sort,
- FT: Fast Fourier Transform,
- CG: Conjugate Gradient,

In terms of classes, we chose specific classes based on the machines. As the Headnode is the less powerful machine, we used the C class for all the benchmarks. We tried to use the D class, but because the complexity gap was too high, the benchmarks took too long to run. For the Asus and Node03 machines, we used the D class for all the benchmarks except for the IS benchmark, where we used the C class. This exception is because the IS benchmark could not compile with the D class. As the time varies for each benchmark and class, we cannot give a precise time for the experiment. In our different tests, the total time of the test ranged from 15 minutes to 30 minutes, depending on the machine used.

3.3.3. Stress test

The two previous experiments are well-known and often used in the literature, but we also wanted to have a more controlled and customizable workload to test the tools. So, we decided to create our stress script based on two different commands: `iperf3` and `stress-ng`.

The first command, `iperf3` [11], is a tool to measure the maximum bandwidth on a network with various available parameters to be set, like the number of parallel streams, the size of the buffer, the time to run, and the protocol to use. To be able to make it run, two machines are needed: one as the server and one as the client. The client is the machine we are monitoring, and the server is just the machine where the data is sent. In our case, the server was always the Headnode, except when it was the machine we were monitoring. As all the servers are connected to a 10Gbps network and a 100Gbps network, we did two tests during each experiment. Before creating the script, we did a small test to see the best number of parallel streams to use, and we found that it was 6 for both tests.

The second command, `stress-ng` [19], is a useful tool that can generate a lot of different types of stress on a machine with many different parameters. It contains a wide range of programs, and each of them can generate stress on a specific part, or multiple parts, of the system. It is even possible to combine multiple programs, either in parallel or in sequence. Those programs can be used to generate stress on the CPU, memory, network, disk I/O, etc. In our test, we used the following configuration:

- A quick CPU stress with a load of 100% and using as many workers as there are cores on the machine.
- Three CPU stress but with a load of 90%, 75%, and 50%, still using as many workers as there are cores on the machine.
- Two CPU stress with a load of 100%: the number of workers was, respectively, half or a sixth of the number of cores on the machine.
- One AIOL stress, which creates multiple asynchronous I/O writes.
- One HDD stress, which continually writes, reads, and removes temporary files, with a size set to 10GB.
- One MEMRATE stress, which exercises the memory with various rates of read and write, using a memory buffer size of 10GB.
- One UDP stress, which transmits a lot of UDP packets.

To accommodate some of our needs, we created two versions of the script: an extended version and a quick version. The extended version runs each `stress-ng` test for almost 1 hour 50 minutes, except the two first CPU stress that uses a load of 100% and 90% that runs for 10 minutes and 1 hour 20 minutes respectively, and the `iperf3` tests for 1 hour. For the quick version, all the tests are run for 10 minutes. Both versions have a sleep of 2 minutes between each test. As they are 12 tests, the extended version is around 20 hours long, and the quick version is 2 hours long.

3.3.4. Overhead test

One of the main concerns when using those types of tools is the overhead they can generate. However, the previous experiments only gave us a comparison of the accuracy of the tools compared to the PDU and each other. To gain insight into their overhead, we created another script that ran a particular workload multiple times: once with any of the tools and once without any tool. We used the workload from the command in the stress test, `stress-ng`. We estimated that setting the load on all cores to 100% would generate too much power consumption, and we may not be able to see the overhead of the tools. So, instead, we set the load to 50% on all cores. Each run of the workload was 10 minutes long, with some time to start and kill the power monitoring tool before and after their respective run. We also added a sleep of 2 minutes between each run. As the frequency of data collection for some of the compared tools can be modified, we run two tests for each of these tools, one with a quick interval of 1 second and one with a longer interval of 5 seconds.

3.3.5. Metrics and graphs

To evaluate the tools, we used different metrics that are commonly used with regression problems. In our case, the predicted values ($\hat{y}$) are the estimated power consumption provided by the tools, and the actual values (y) are the power consumption measured by the PDU. In the formulas, n is the number of samples and $\bar{y}$ is the mean of the actual values.

The first metric we used is the Mean Absolute Error (MAE). As the name suggests, this metric is the average of the absolute differences between the predicted values and the true values. It is one of the most straightforward metrics to use and understand and gives a good idea of the error of the model. Of course, the lower the value, the better the prediction.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

The second metric is the Rooted Mean Squared Error (RMSE). This metric is the square root of the average of the squared differences between the predicted values and the true values. Compared to the MAE, the RMSE penalizes large errors more because of the square operation. So, this metric will give us an idea of the outliers in the prediction.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

The third metric is the R-squared score (R^2). This metric is a statistical measure that represents the proportion of the variance for a dependent variable that is explained by an independent variable. So, it is a measure of how well the predicted values fit the actual values. It is a value between 0 and 1, where 1 means that the predicted values perfectly fit the actual values. It will also give us an idea of how well the tools are performing on the different machines as the power consumption from one machine to another is different, and thus, the other metrics will, of course, vary as well and will not be directly comparable.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

For each test, we will present the results similarly. We will first show, for each machine, a graph of the estimations from the three tools on top of the PDU measurements. Those plots allow us to visually compare the estimations of the tools to the actual power consumption. It will be followed by a box graph showing the error of each tool compared to the PDU measurements and a table with a summary of the above metrics to give a more detailed view of the results. As our goal is to see the accuracy of the estimation for a running application and that some of our tests had sleep time between each test, we removed these parts before generating the box graph and computing the metrics. This way, we only keep the data when the application is running.

3.3.6. Data processing

When the experiments are finished and the files are collected, some processing is needed to compare the tools. First, we created a new column with the time in seconds since the beginning of the experiment. This makes it easier to plot the data. Then, because of the behavior of some of the tools, the estimations sometimes get way too high just for some steps. This makes some of our metrics irrelevant, such as the R^2 value, which went far below 0. We estimated that those values could be easily discovered at runtime, so we decided to remove those values before computing the metrics. We still kept the values in the graphs to see those behaviors, but we did not always include them in the range of the y-axis as they could make the graph unreadable. For PowerJoular, some of the data we got was ”NaN” or ”Infinity”. Those values were set to 0 as they only happened when none of the processes were running.

3.4. Summary

In this chapter, we presented the different machines we used for the experiments, the configurations for the tools we chose to compare, and the experiments we ran to evaluate the tools. We also presented the metrics we used to evaluate the tools. In the next chapter, we will show the results of the experiments, discuss them, and try to answer our first two research questions.

4 Experiments

Now that we have described our setup, we can present the results of our experiments. Each experiment was run at least three times on each machine and with the three chosen tools together. To make the visualizations clearer, only one run is shown for each experiment. As stated in the previous chapter, the time of each run can vary from 15 minutes up to 20 hours, depending on the test running. In addition, more experiments were run to check the correct behavior of the tools and the workloads, verify the parameters used, and fix some issues that were primarily due to wrong configurations of the tools or the machines. So, the different experiments were run over several weeks. It is also worth noting that for the SPECPower test, we only ran the test on the Headnode and the Asus machines since the benchmark was not installed on the Node03 machine. Each of the following sections will present the results of one of the tests. We ordered the sections in the same order we presented the tests in the previous chapter: the SPECPower test, the NAS Parallel Benchmarks, the Stress test, and finally, the Overhead test.

At the end of this chapter, we will have the answers for our first two research questions: *What are the existing software-based power meters, and how do they work and compare in terms of accuracy, performance, and features?* and *What are the best techniques and metrics to use to estimate the energy consumption of an application?*.

4.1. SPECPower test

Let us start with the SPECPower test. As explained in the previous chapter, the SPECPower is composed of different workload phases, going from full load to idle. The idea of this test is to see how the tools react to varying levels of CPU usage.

The results of the SPECPower test are shown in Figure 4.1. Looking at both graphs, we can already make some interesting observations. The first thing we can easily see is that, even if their estimations are not perfect, all three tools can follow the general trend of workload changes. We can easily see when the phases of the workload change and the decrease between each phase. This is already a good sign as it shows that the tools are correctly measuring the power consumption of the application. But before looking in more detail at their trends, we can spot some behaviors worth noting.

The first behavior we saw was the measurements from the PDU. When none of the tools are running, the PDU measurements are not at a constant zero but have minor variations. The reason is simple: in the previous chapter, we assumed that only the experiments were running on the machines, but of course, other essential services were running, which caused those variations. It is also important to remember that as the raw measurement is about the whole system, we substrated

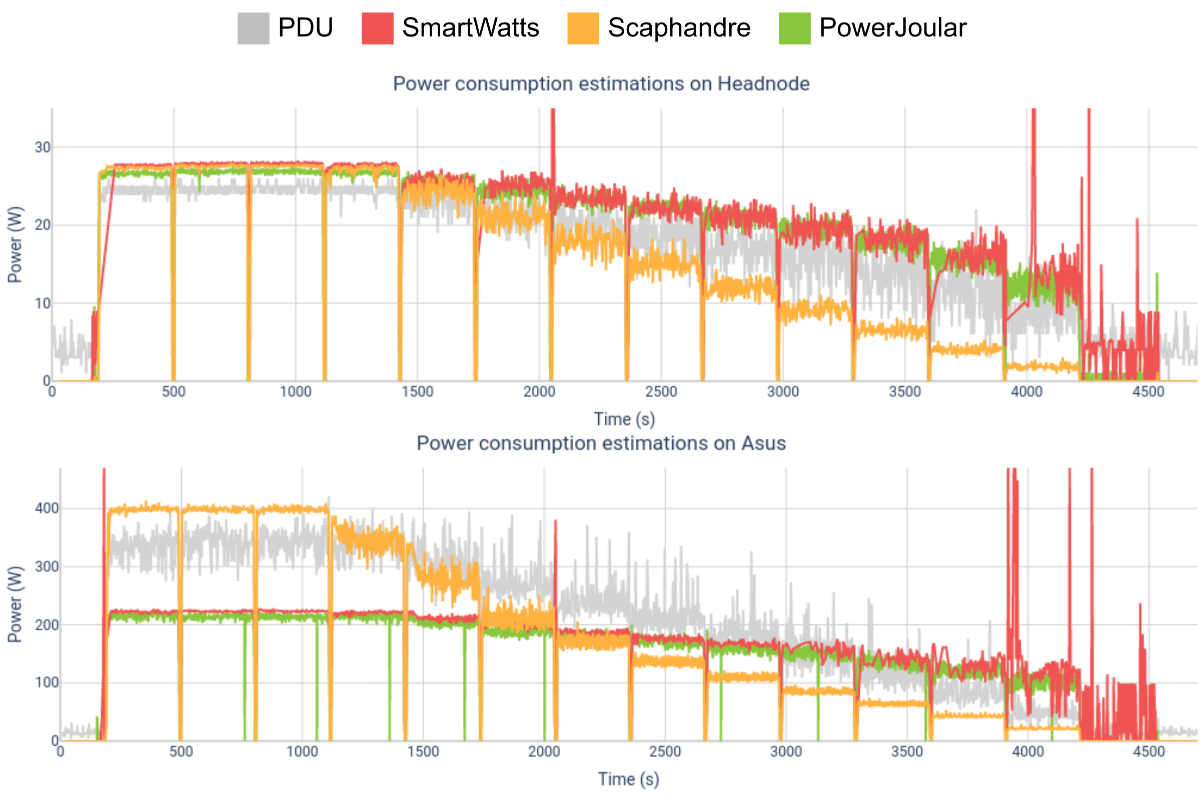


Figure 4.1: Estimations for the SPECPower test.

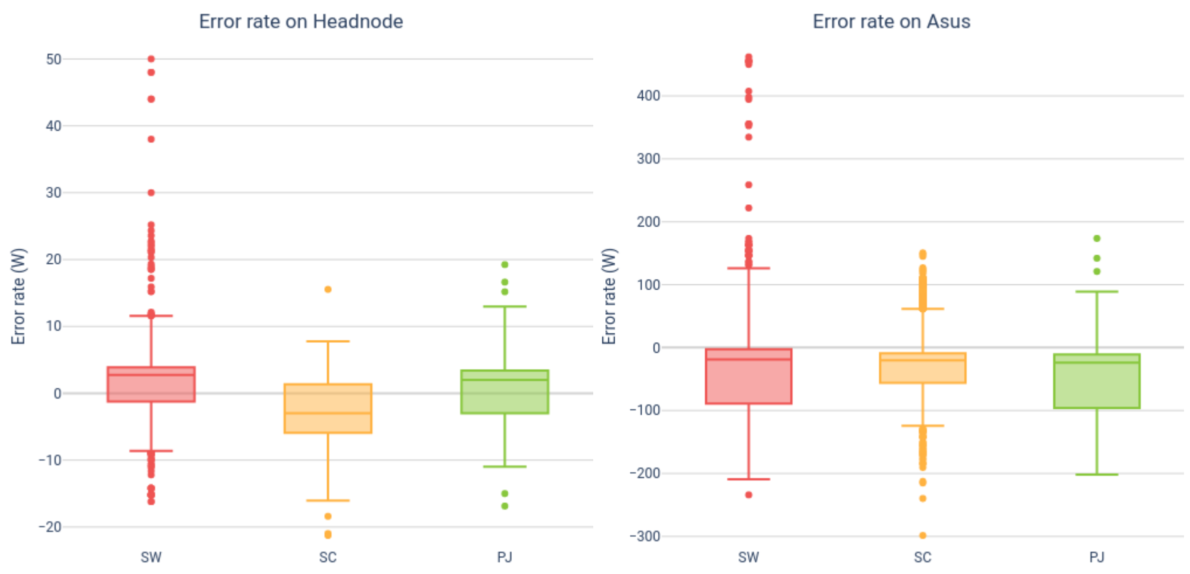


Figure 4.2: Error rate of the estimations on the SPECPower test.

the lowest value from the rest of the measurements to get the power consumption of the running application.

Another noticeable behavior is seen in the SmartWatts tool. We can see that at the beginning of some workloads, it seems that the first estimations are missing or not correct before it stabilizes. We can also see some major spikes at some points, like the start of the first phase on the Asus machine. The reason for this behavior is because of how the tool works internally to make its estimations. As explained in the previous chapters, SmartWatts trains different regression models on different data buffers based on the CPU frequencies. So when the CPU frequency changes enough to make the tool switch to another model, if the model is not already trained, the tool will have to wait for the minimum number of samples, as defined in the configuration, to train it. This idea explains why there are some missing estimations at the beginning of some phases, but for the spikes, we need to go deeper. Even if the tool has the required number of samples, the model might not have enough data to be accurate, or the counters might not be stable enough for some CPU frequencies. However, the tool has a mechanism to retrain the model when he found that the estimation was not good enough. So spikes are undoubtedly due to the model not being accurate enough, but the tool corrected it for the successive estimations.

In a similar way, we can see that the estimations of PowerJoular drop to zero at some points on the Asus machine. While we do not have a clear explanation for this behavior, one possible cause would be that the overhead of all the tools running together with the low interval of 1 second for the tool makes it miss some estimation.

Now let us look at the estimations of the tools. On both machines, we can see that, even if SmartWatts has spikes and missing values at some times, it generally follows the same trend as PowerJoular, compared to Scaphandre, which has a different trend. On the Headnode, both estimations are almost always above. Their estimations also begin to vary more and more as the intensity decreases. On the Asus machine, their estimations for the first workloads are quite low compared to the PDU. It is only around phase 9 that their estimations and the PDU measurements are close before the estimations go above the measurements until the second last phase. The estimations are below the measurements for the last phase. For Scaphandre, the estimations have a similar trend on both machines. The estimations are consistently above the measurements for the first phases, and then around phase 5, the estimations go below the measurements.

Figure 4.2 gives us a good view of the performances of the tools during the test. For the Headnode, all three tools have their interquartile range (IQR) across the 0W line. However, SmartWatts has the biggest part of its IQR and its median above the line, while Scaphandre has them below it. PowerJoular is the only one with its range almost centered around the 0W line. On the Asus machine, all three tools have a similar plot, with their IQR just below the 0W line. We can see that the IQR and range of Scaphandre are much smaller than the two other tools. Both graphs, we can see that SmartWatts has the most outliers, which is consistent with the spikes and missing values we talked about earlier.

Finally, looking at the metrics in Table 4.1, we can see that the estimations of PowerJoular are the most accurate on the Headnode while Scaphandre is by far the most accurate on the Asus machine. Those results are consistent with what we saw in the previous paragraphs.

	SmartWatts			Scaphandre			PowerJoular		
Machine	MAE	RMSE	R^2	MAE	RMSE	R^2	MAE	RMSE	R^2
Headnode	3.991	5.209	0.906	4.288	5.070681	0.911	3.602	4.149	0.940
Asus	60.615	80.733	0.857	44.228	54.475	0.935	57.868	74.970	0.876

Table 4.1: Metrics of the estimations on the SPECPower test.

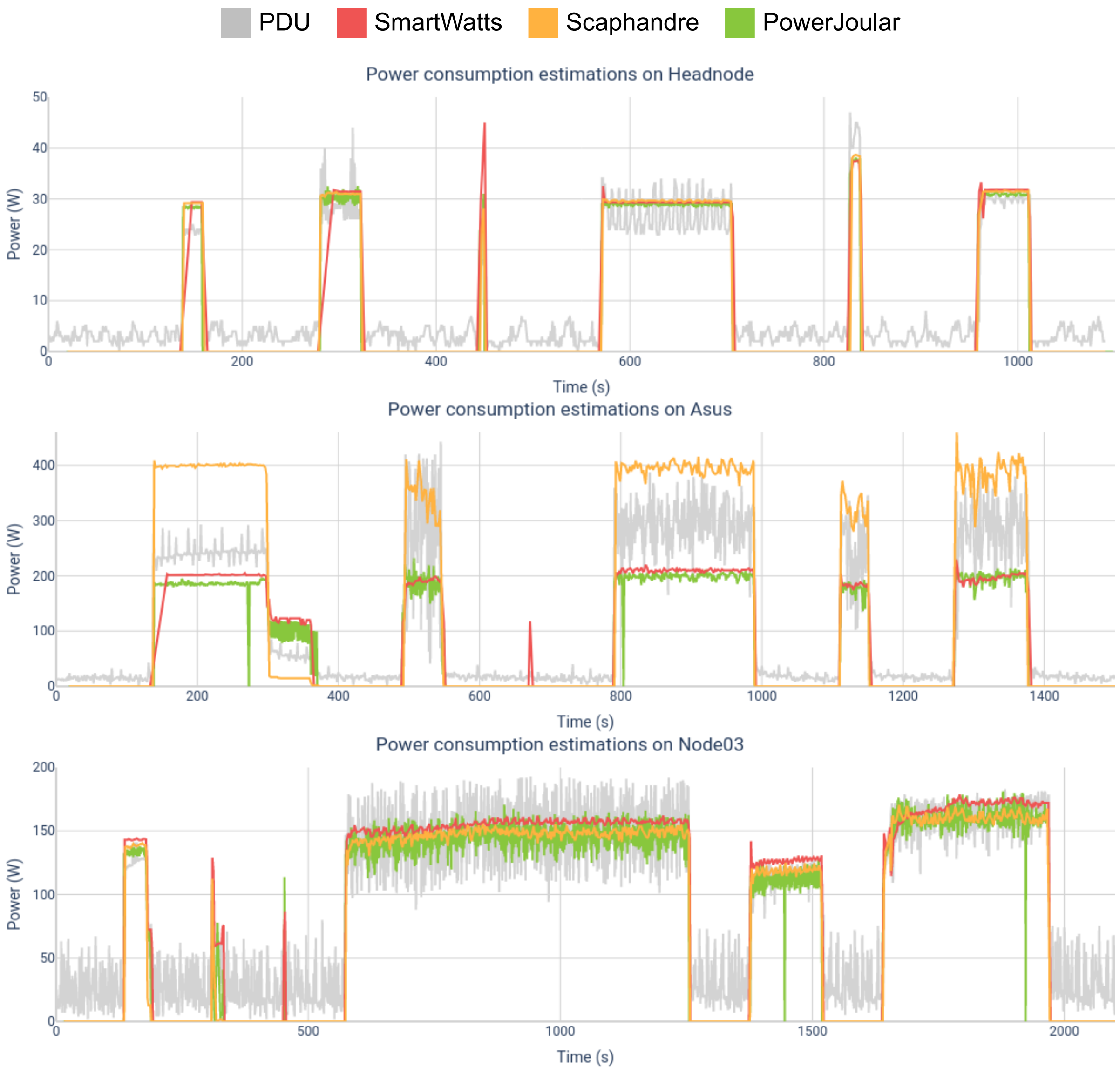


Figure 4.3: Estimations on the NAS Parallel Benchmarks.

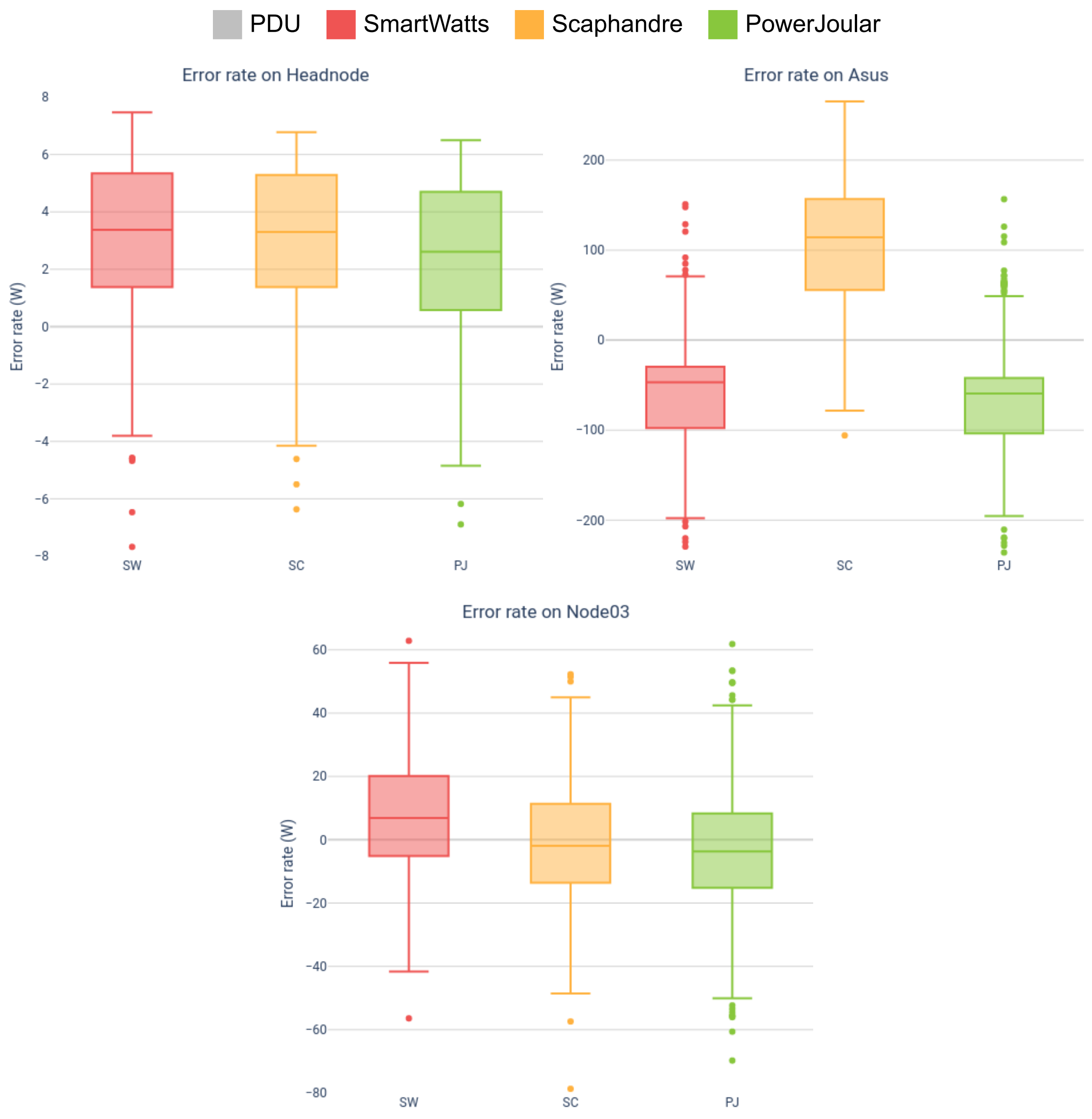


Figure 4.4: Error rate of the estimations on the NAS Parallel Benchmarks.

4.2. NAS Parallel Benchmarks test

The first thing we can see in Figure 4.3 is the time it took to run the different benchmarks on each machine. This is mostly interesting for the Asus and the Node03 machines as they were using the same classes for the benchmarks, and we can see that on the Node03, it took around 10 minutes more than the Asus. Another thing is that the IS benchmark was too short to get any meaningful data from it. So, while we kept it in the power consumption graphs, we did not include it in the box plots and the metrics.

Still looking at the graphs, we can see that the estimations of the tools are close to each other and to the measurements on the Headnode and the Node03 machines. However, on the Asus machine, the estimations of Scaphandre are again different from the other tools. It is always above the measurements, while the other tools are below. For the EP benchmark, the estimations of Scaphandre are less accurate as the gap between the estimations and the measurements is more than 100 Watts. Those observations are consistent with what we can see in the box plots of the error rate in Figure 4.4. On the Headnode and the Node03 machines, the error rates of the three tools are similar, while on the Asus machine, the error rate of Scaphandre has its IQR way above the 0W line, while the two other tools have their IQR below it.

In terms of metrics, we can see in Table 4.3 that the estimations of powerJoular are the most accurate on the Headnode and the Node03 machines, closely followed by Scaphandre and then SmartWatts. However, on the Asus machine, the estimations of SmartWatts are the most accurate. Another interesting thing to note is that the metrics of the three tools are close on the Headnode and the Node03 machines.

	SmartWatts			Scaphandre			PowerJoular		
Machine	MAE	RMSE	R^2	MAE	RMSE	R^2	MAE	RMSE	R^2
Headnode	3.748	4.174	0.978	3.604	4.078	0.979	3.122	3.641	0.983
Asus	73.273	85.934	0.891	107.968	121.458	0.781	79.353	91.506	0.876
Node03	16.848	21.361	0.979	16.150	20.808	0.980	15.829	20.490	0.981

Table 4.2: Metrics of the estimations on the NAS Parallel Benchmarks.

The main goal of this test was to be able to compare our results with the two papers that were using the same benchmarks. But when we went back to the papers, we could not find any metrics to compare to. The only thing we could compare was the result of the SmartWatts paper [12]. In this paper, the authors show a box plot of the absolute error rate of the model compared to the reference. The estimations of the model were for the overall power consumption of the machine from the aggregated counters, while the reference was the RAPL value. From the box plot, we can see that the IQR stayed between 0 and 3 watts for all frequencies, with the maximum value being a bit more than 6 watts. This can be compared to our error rate on the Headnode, as it is the machine with the most similar configuration to the one used in the paper. Even if the two experiments are not directly comparable, we can still see that our error rate is close to the one in the paper.

4.3. Stress test

The stress test gives us interesting results. The first thing we can see in Figure 4.5 is the difference between the estimation of the tools on each machine. On the three machines, even with its spikes, SmartWatts seems to be the one that is always close to the measurement. The only exception is during the workload of 100% on the Node03 machine, where its estimations dropped to zero after a third of the workload. This could be due to the overhead of all the tools running together with the workload that made the reading of counters by the tool unstable. To check this hypothesis, we did another run with only SmartWatts and the workload. This test resulted in the same behavior. So this means that the drop is not due to the overhead of the tools but to the workload itself taking too much

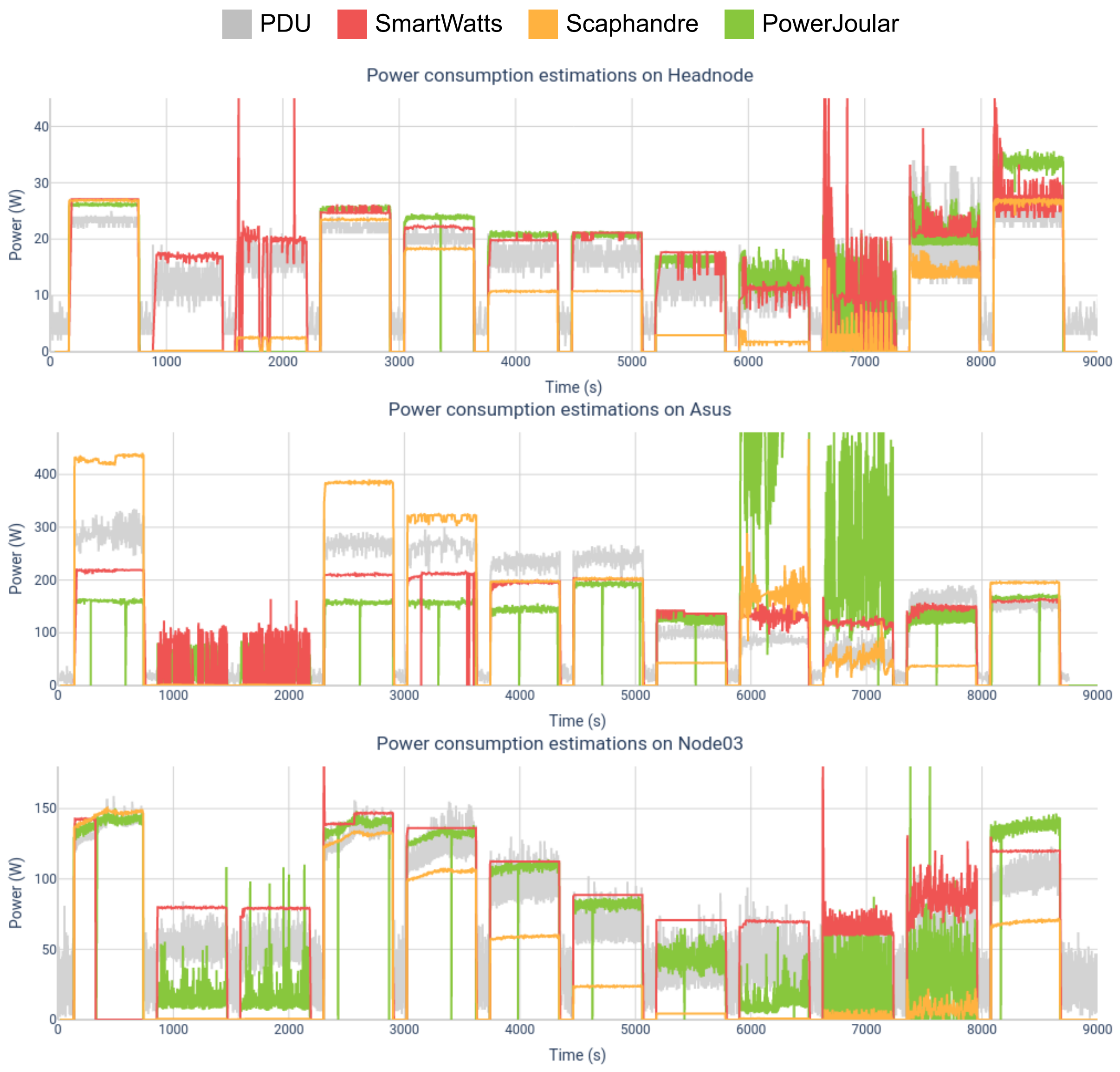


Figure 4.5: Estimations for the Stress test.

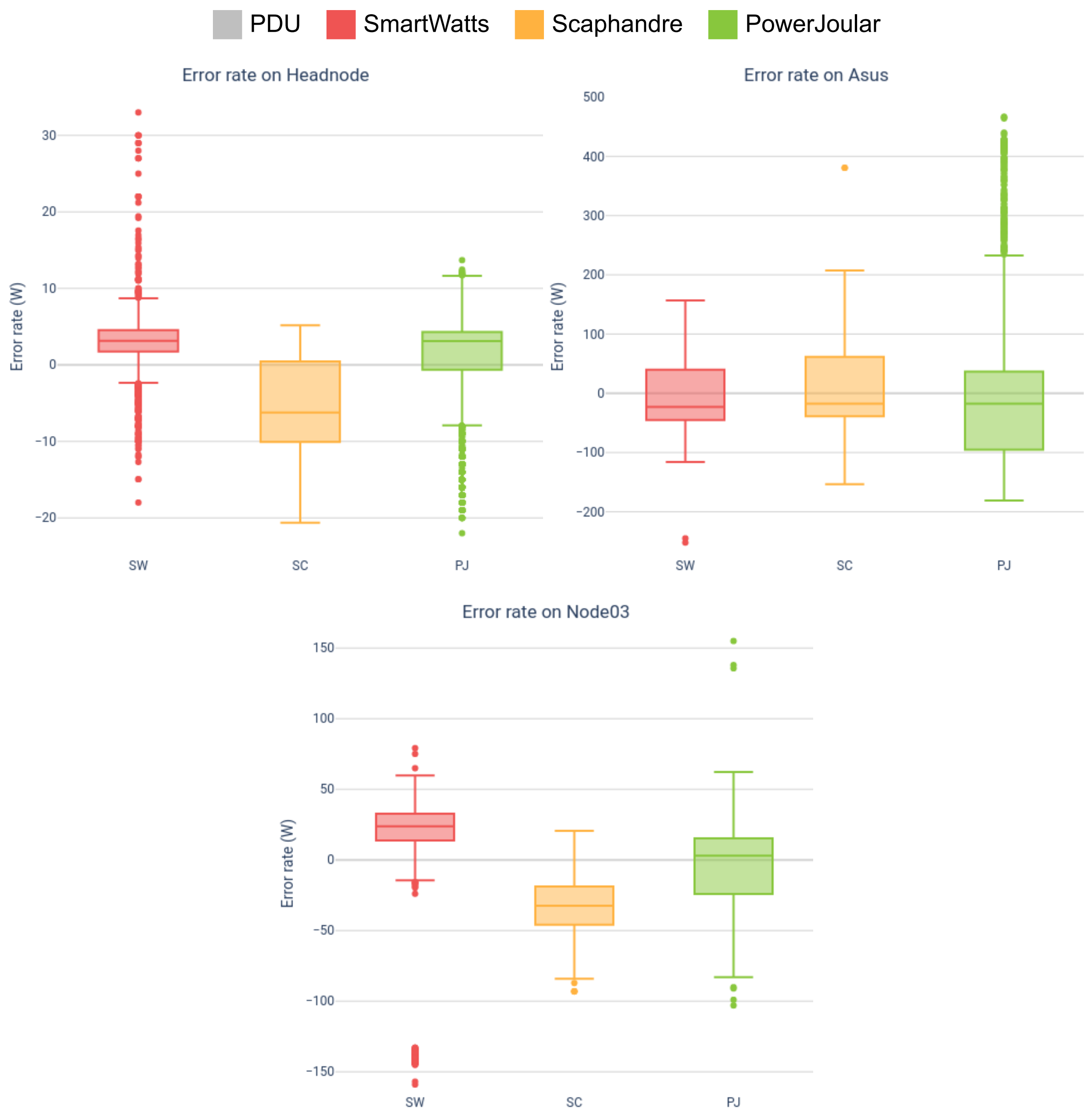


Figure 4.6: Error rate of the estimations on the Stress test.

of the resources of the machine. For the load workloads, the three tools seem to have a similar trend to the SPECPower test for the Headnode and the Asus machine, as the Node03 machine was not part of the SPECPower test. For that machine, while the estimations of SmartWatts and PowerJoular are always above and close to the measurements, the estimations of Scaphandre are most of the time below the measurements. The gap between the estimations and the measurements is more significant as the workload decreases. For the network workloads, using the `iperf` tool, we can see that SmartWatts has a good estimation. The estimations of Scaphandre and PowerJoular, on the other hand, are low compared to the measurements. This is significant on the Headnode machine, where PowerJoular is almost always at zero, and Scaphandre is only slightly above during the second workload. The results vary a lot between the machines for the last four workloads. For example, for the AIOL and HDD stresses, PowerJoular seems to have difficulties estimating the power consumption of the programs. On the Asus, its estimations would even go up to 1600 Watts, which is too high when we see that the PDU measurements are around 100 Watts.

If we now look at the error rate in Figure 4.6, we can see that SmartWatts always has the smallest IQR and range on the three machines, but it still has many outliers on the Headnode and the Asus machines. For PowerJoular, most of its outliers are due to its wrong estimations during the network workloads on the Headnode, as well as the AIOL and HDD workloads on the Asus machine. But, it is the only tool with its IQR around the 0W line on each machine, with even its median close to zero on the Node03 machines.

The metrics in Table 4.3 show that the estimations of SmartWatts are the most accurate on the Headnode and the Asus machine. However, the estimations of PowerJoular are the most accurate on the Node03 machine. We can also see the low performance of PowerJoular on the Asus machine, with an R^2 of 0.362, caused by the wrong estimations during the AIOL and HDD stresses.

	SmartWatts			Scaphandre			PowerJoular		
Machine	MAE	RMSE	R^2	MAE	RMSE	R^2	MAE	RMSE	R^2
Headnode	3.766	4.676	0.930	6.631	7.942	0.798	5.887	7.407	0.825
Asus	43.604	48.650	0.931	63.448	76.846	0.827	99.503	147.550	0.362
Node03	31.208	42.733	0.746	32.446	37.104	0.809	21.294	26.462	0.903

Table 4.3: Metrics of the estimations on the Stress test.

4.4. Overhead test

In terms of power, the overhead from the tools is negligible. To compute the overhead, we took the average power consumption of the machine with the workload running but without the tools as the reference. Then, we did the same for each run and calculated the percentage of the difference with the reference. The results are shown in Table 4.4. The most significant overhead is from Scaphandre (1s), with 0.961% more than the reference. The one with the least overhead is SmartWatts (5s), with only 0.707% more than the reference. SmartWatts (1s) also has a low overhead, which makes SmartWatts the tool with the least overhead. There is also an increase when using 1s intervals compared to 5s intervals, which is logical as the tools need to do their estimations more often. The overhead of PowerJoular is also high, close to the one of Scaphandre using the same interval of 1s. Overall, the overhead of the tools is still below 1%, so there is no significant impact on the power consumption of the machine.

To confirm the results, we used the metrics arguments available with `stress-ng` to get the statistics of the workload running with and without the tools. The interesting part of those statistics can be seen in Table 4.5. In terms of real-time, we can see that the values only vary by 0.01 milliseconds between the runs with and without the tools. The combined user and system time also does not differ much between the runs. However, we can see that the user time tends to decrease, and the system time tends to increase when the tools are running. The most insightful metric is the

No tools	0
SmartWatts (5s)	+0.707 %
SmartWatts (1s)	+0.839 %
Scaphandre (5s)	+0.861 %
Scaphandre (1s)	+0.961 %
PowerJoular	+0.927 %

Table 4.4: Percentage difference of the average power consumption

bogus operations per second. As explained in the documentation, this metric is not intended to be a scientifically accurate benchmarking metric, but it can give a rough idea of the throughput of the workload. In our case, we can see some differences between the runs. The run without the tools has the highest throughput, averaging 856.41 bogus ops per second. The run with Smartwatts (5s) has the lowest throughput, with an average of 853.60 bogus ops per second. The other runs have an average throughput of around 855 bogus ops per second. Surprisingly, the SmartWatts (1s) has a throughput of 855.53 bogus ops per second, which is the closest to the reference.

	Real-time	User time	System time	User + System Time	Bogus ops / s
No tools	600.01	1800.40	0.80	1801.20	856.41
SmartWatts (5s)	600.02	1800.08	0.74	1800.82	853.60
SmartWatts (1s)	600.01	1799.94	0.96	1800.90	855.53
Scaphandre (5s)	600.01	1799.75	1.07	1800.82	854.99
Scaphandre (1s)	600.00	1799.79	1.06	1800.85	855.14
PowerJoular	600.02	1799.91	0.92	1800.83	855.26

Table 4.5: Metrics from the `stress-ng` tool for the overhead test.

4.5. Discussion

Now that we have presented the results of our experiments, we can answer our first two research questions.

What are the existing software-based power meters, and how do they work and compare in terms of accuracy, performance, and features? In the second chapter, we have presented different software-based power meters or tools that can be used to build one. We also explained how they work and what their main features are. At the end of the chapter, we have chosen three tools to compare their performance during our experiments: SmartWatts, Scaphandre, and PowerJoular. They were chosen on top of the others because they were the most in line with our main objective since they enable us to estimate energy consumption at a process level.

All three tools gave quite good estimations of the power consumption of the applications during our experiments and have a relatively low overhead. However, we have seen that the estimations of the tools can vary a lot depending on the machine and the workload. For the SPECPower test, the estimations of PowerJoular were the most accurate on the Headnode, while Scaphandre was the most accurate on the Asus machine. For the NAS Parallel Benchmarks, the estimations of PowerJoular were the most accurate on the Headnode and the Node03 machines, while SmartWatts was the most accurate on the Asus machine. For the Stress test, the estimations of SmartWatts were the most accurate on the Headnode and the Asus machine, while PowerJoular was the most accurate on the Node03 machine. Overall, the estimations of PowerJoular were the most accurate on the Headnode and the Node03 machines, while SmartWatts was the most accurate on the Asus machine.

What are the best techniques and metrics to use to estimate the energy consumption of an application? Looking at the results of our experiments, there is no unique winner to this question. Overall, both SmartWatts and PowerJoular are the most accurate tools, but if we look at each workload individually, the previous statement cannot be generalized, as both of them are sometimes outperformed by Scaphandre. If we take aside the accuracy of the tools, then Scaphandre is the most consistent, with the least spikes or missing values, and thus the least outliers. Another detail shows that there is no clear answer to this question. While the technique used by PowerJoular is quite similar to the one used by Scaphandre, they both use a simple power formula based on the process CPU usage, the technique used by SmartWatts is completely different, with its regression models. This detail goes against the observations from our experiments, as most of the time SmartWatts and PowerJoular were following a similar trend that was different from Scaphandre. As there is no best technique among the three tools, with each of them having their strengths where the others have weaknesses, the best technique would be to combine them and bring the best of each tool. This would improve the estimations and help to make them more stable, as it would be possible to detect when one of the tools is going wrong.

In terms of metrics, as there is no best tool, all the metrics used by those tools are useful to make the best estimations. They all have one metric in common: they all use the available RAPL values to make their estimations. In addition, Scaphandre and PowerJoular use the CPU usage of the processes in their computation. For SmartWatts, it retrieves some of the HwPC counters for each of the processes.

In the next chapter, we will try to mix the three tools to see if we can make a more accurate software-based power meter and answer our last research question.

5 The Geryon program

In the previous chapter, we compared the performance of three different programs and answered the first two research questions. We found out that there was not a clear winner between the three programs in terms of accuracy and performance. We then proposed that mixing the three tools would be an excellent solution to improve the power consumption estimation. In this chapter, we will present the design of the hybrid program we implemented to test this idea. We name this new program Geryon[1], drawing inspiration from the three-bodied giant from Greek mythology. In the next chapter, we will present the results of this new program and answer the last research question.

As all the programs we used are open-source, we were able to look at the code and understand how they work internally. We then used this knowledge to implement the new program. The idea was to select one of the programs as the base and add the features of the other two programs.

5.1. The base program

We do not want to recreate a new program from scratch, as it would take time and resources. Instead, we will use one of the programs as the base and add the features of the other two programs. Many variables have influenced the choice of the base program, like the language it is written in, the ease of use, the performance, and the features it already has.

After considering this, we chose the Scaphandre program as the base of our Geryon program. There are a few reasons for this choice. The best way to explain why we decided to use Scaphandre is to explain why we did not choose the other programs. SmartWatts is a good program, but it needs two binaries to work: the sensor (hwpc-sensor) and the formula (SmartWatts). While the formula is written in Python, the sensor is written in C, and there is a third code, the PowerAPI, written in Python that is used as a library by the formula to be able to read the output of the sensor. This division makes the program more complicated to use and understand. On the other hand, PowerJoular is a single binary, but it does not have that many features, as we cannot even change the interval of the estimations. Moreover, it is written in Ada, which is not a widely used language. Compared to those two programs, Scaphandre is a single binary that can be run on most machines. It retrieves the metrics per process instead of using the CGroup, so it does not require any particular manipulation of the system. It already retrieves many other metrics on the CPU and the host system and has a solid codebase. Finally, it is written in Rust, which comes with a good repository of valuable libraries, like some for machine learning, which will be needed to implement the regression feature.

[1]The source code of the Geryon program is available at `https://github.com/WhyW0rry/Geryon`

5.2. The time ratio

Now that we have chosen a base program, we can add the features of the other two programs. The first feature we will add is the time ratio calculation from PowerJoular, which is the simplest one to implement as most information is already available in Scaphandre. While the idea used by Scaphandre in its estimation is similar, the difference is that Scaphandre directly retrieves the CPU utilization of the process from the system by using the `sysinfo` crate. On the other hand, PowerJoular calculates the time ratio, which is the ratio between the time a process has been running and the time the CPU has been busy.

To implement the feature, we first needed to modify the `mod.rs` for the sensors to include all the metrics we needed. In the initial version of Scaphandre, a `stat_buffer` is used to store the CPU statistics for each socket. Those statistics are the different times from the CPU that are retrieved from the `/proc/` file system. They are stored in a `CPUStat` struct that already implements a method to calculate the total time jiffies, which correspond to the total time the CPU has been busy. We added a new method to calculate the total time the CPU has been running by including the idle time. Then, we added a method on the socket structure to calculate the time difference between the two last statistics stored in the `stat_buffer`. This gives us the `ctotal` and `cbusy` we need for our calculation. The last addition to the file is a new refresh method, used alongside the others, that uses the new methods to retrieve the CPU times for each socket and compute their sum. It also retrieves the power consumption before giving all this to the `ProcessTracker`.

In the `ProcessTracker`, we added a new method that iterates through all the processes. For each process, it will retrieve the `stime` and `utime` from the two last records of the process, stored at each timestep, and calculate their difference. At this point, we have all the information we need to do the calculation used in PowerJoular. We used the formulas stated and explained in Section 3.2.4. The result is stored on the last record of the process. Some minor changes were made to the exporters to include the new metrics in the output.

5.3. The regression

With the time ratio implemented, we can now add the regression feature from SmartWatts. There will be a difference with the original SmartWatts program, as we will not use CGroup to group the processes. We will compute the estimations for each process individually to stay consistent with how the Scaphandre program works.

The first step in implementing the idea of regression was to find a way to initiate and read the HwPC counters. We used the `perf-event2` crate, which provides a wrapper around the Linux `perf_event_open` API. The chosen counters were similar to the ones used by SmartWatts. The first two counters are generalized events, so we used the available `PERF_COUNT_HW_CACHE_MISSES` and the `PERF_COUNT_HW_INSTRUCTIONS` configurations from `perf_event_open` to access them. For the rest of the counters, we had to find their specific event number based on the CPU model. For the Intel CPUs, we added the `CPU_CLK_UNHALTED:REF_P` (code `0x3c`) and the `CPU_CLK_UNHALTED:THREAD_P` (code `0x3e`) counters. For the AMD CPUs, we added the `CYCLES_NOT_IN_HALT` (code `0x76`) and the `RETIRED_OPS` (code `0xc1`) counters. All those counters were hardcoded to the source code, as we did not need to change them during the execution of the program. We used the vendor ID of the CPU to determine which counters to use. When looking at the crate documentation for the counters, we found out that it was possible to directly retrieve the sum of the counters per CPU core for all PIDs instead of doing it for each PID and then summing them.

Like with the time ratio feature, the first file that needed to be modified was the `mod.rs` for the sensors. First, we added some code to the `CPUCore` structure to create and store the counters when a new core is added to the topology during the initialization of the sensor. We also modified the `CPUSocket` structure to retrieve and sum the counters of their cores at each time step and store them in a buffer list with the RAPL value. Then, similarly to SmartWatts, when there are enough metrics in the history buffer, an Elastic Net regression model is trained, and a prediction for the socket is computed. As SmartWatts was written in Python, it used the `scikit-learn` library to train the model. This library is not available in Rust, so we had to find an alternative. We chose to use the `linfa` library, a machine learning library for Rust with a similar spirit as `scikit-learn` and focus on common preprocessing tasks and classical algorithms. This library is not as complete as `scikit-learn`, but it has an implementation of the Elastic Net regression model through the subpackage `linfa-elasticnet`, which is what we need. The model and the socket prediction are then passed to the `ProcessTracker` to compute the prediction for each process.

Then, the socket prediction is compared with the RAPL value, and the model is retrained if needed. For its side, the `ProcessTracker` was modified to create and read the HwPC counters of each process PID at each refresh. To limit the number of counters, we only retrieve the counters of the process that match the given regex. If a model is available, the `ProcessTracker` will use it to predict the power consumption of the process. If the model is not yet trained, the `ProcessTracker` is still used to create the HwPC counters and set the prediction value at 0. We also added some arguments to the exporter to allow the user to specify the TDP of the CPU, together with the minimum number of samples needed to train the model and the maximum number of samples to keep in the history buffers.

After the first experiments with this implementation, we stepped into an issue with the counters of the CPU cores. While everything was working correctly on the Headnode machine, some counters were missing when running the program on the Asus one. This issue was caused by the fact that each CPU has a limit to the number of counters that can be created, and as the Asus machine has a lot more cores, the limit was quickly reached. To fix this, we added an alternative way to recover the counters for the socket, which will sum the counters of each process instead of the cores as the SmartWatts program does. This new way is activated by a boolean variable in the sensor that the user can change. By using this feature, the number of counters created is reduced. However, it also means that all running processes need to be monitored; otherwise, the sum of the counters will not accurately represent the socket usage.

5.4. The external model

One way to make the model more accurate would be to train the model on more data instead of just the last data. Another way to improve the accuracy would be to have a way to train and test different types of models, and then just plug the best one into the program. To allow those two features, we modified the code so that anyone could train and use an external model. Having an external model would also allow us to use the PDU values as the target, as this would include all the power usage, even those not taken into account by the RAPL value. To do so, we had to add two things in our code: a way to export the metrics such that they could be easily read by another program while not taking too much space and a way to import a model and use it in the program.

For the metrics, we decided that the best way to export them would be to use CSV files for each level of metrics: the host, the sockets, and the processes. So, we created a new exporter that would be used to write all the metrics in the three CSV files. For the host, we write the RAPL values and the CPU frequency. For the sockets, we write the RAPL values and the sum of the HwPC counters from each core. For the processes, we write the HwPC counters, the process CPU utilization, and some other metrics that we thought could be useful, like the estimation from Scaphandre and the estimation from the newly implemented time ratio.

For the import of the model, we decided to use `Tensorflow` [26] to load a trained model into the program. The reason we chose `Tensorflow` is that it is widely used in the machine learning domain and that it comes with a rust-binding package that allows us to easily load the trained model and use it to make a prediction. Adding the model loading in the code was relatively straightforward. We added a new argument to the program to specify the path to the model. Then, we used the `Tensorflow` crate to load the model, create the input tensors from the required metrics, and finally make a prediction in the `ProcessTracker`, as we did with the previously implemented internal model. To make it easier and avoid any issues, when an external model is used, the program will not train any internal model, and the regression prediction will be made only with the external model.

Now that we have a way to retrieve the metrics and import an external model, we need to create and train a model. To do so, we created a new script written in Python, as we need to use the `Tensorflow` library.

First, the `pandas` library [34] is used to read the CSV files and the text file containing the values from the PDU and make a table for each file. In each table, rows that have missing values or do not have a valid timestamp are dropped. For the processes table, a new table is created containing the aggregated version of the data: for each distinct time step in the original table, we group the row and sum the values. The next step was to create a unique table containing all the information of the other tables. To do so, the tables were merged based on the timestamps. To ensure that any dropped row of one table would not impact the rest, we used an inner join: if there is a missing timestamp in one of the tables, that timestamp is also discarded in the resulting table.

When the merged table is created, an analysis is done on it. A correlation matrix is computed to see which information correlates to the values from the PDU and the RAPL values. We saw that 5 of the six HwPC metrics we retrieved were strongly correlated with the PDU values and even more with the RAPL values. This is logical as the PDU values represent the consumption of the complete system, including parts that are not taken into account by the RAPL values or the HwPC counters. In the other metrics, we can see that the CPU usage of the processes is also correlated with the PDU values and the RAPL values, but less than the HwPC counters. Without surprise, the estimation from Scaphandre and the estimations from the time ratio are also correlated with the PDU values and the RAPL values.

The last step was to create and train the model. To do so, we used the `Tensorflow` library and its `Keras` [6] API to build the model. We created a simple model with two hidden layers of 64 neurons each and a ReLU activation function. We then trained the model on 80% of the data and tested it on the remaining 20%. We used the mean squared error as the loss function and the SGD optimizer with a learning rate of 0.0001. The same metrics we used during our experiments were used to evaluate the model. We used a large number of epochs to ensure that the model was well-trained, using 10% of the training data as a validation set and an early stopping to stop the training if the loss on the validation set did not improve for 10 epochs. After the training, we saved the model to a format that could be loaded by the modified Scaphandre program.

5.5. The complete program

Now that we have implemented all the features we wanted to add to the program, we can present the complete program as it is now. With Scaphandre as its base, the program is written in Rust and is composed of two parts: the sensors and the exporters. When the user starts the program, it must choose which exporter to use. If he chooses the `collect` exporter, the program will start the sensors, collect the metrics, and store them in a CSV file. If he chooses the `JSON` or the `stdout` exporters, then the three estimations will be computed: the estimation from Scaphandre, the estimation using the time ratio, and the estimation from the regression model. If the user wants to use an external model, he can specify the path to the model, and the program will use it instead of training an internal model. When the exporter is chosen and started, the exporter will initiate the correct sensor based on the platform.

During the initialization, the sensor will retrieve the topology of the system and initiate the HwPC counters for the CPU. Then, at each time step, it will first compute the estimation from Scaphandre. To do this, it retrieves the CPU utilization of the process from the system and multiplies it by the RAPL value. Then, it will retrieve the busy time and total time of the CPU. For each process, those values will be used to compute the time ratio, which will be multiplied by the RAPL value to get the estimation. Then, still at each step, the sensor will retrieve the counters and the RAPL values and store them in a buffer. At the same time, it tracks which processes are running and their resource utilization. If a new process matches the given regex, the sensor will create the HwPC counters for this process. For the ones with already created counters, the sensor will sum them to have the total counters for the socket. Those summed counters can replace the counters of the cores in the buffer if the user chooses not to activate them due to the limited number of counters. If enough samples are in the buffer, the sensor will train a model if none is already trained or given. It will then use the model, either the trained one or the external one, to predict the power consumption of each process that matches the given regex. The last step, if the internal model is used, is to predict the power consumption of the socket using the summed counters and compare it with the RAPL value. If the difference is too high, the model is retrained. All those estimations are given to the exporter, which will write them in the output file or print them on the standard output.

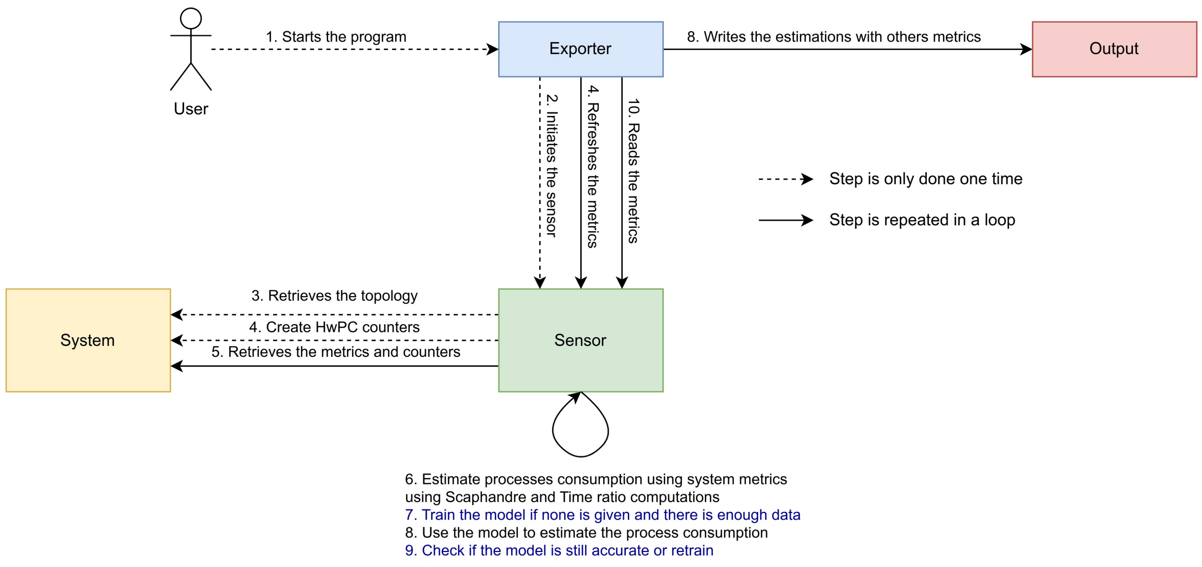

User
1. Starts the program
Exporter
8. Writes the estimations with others metrics
Output
2. Initiates the sensor
4. Refreshes the metrics
10. Reads the metrics
Step is only done one time
Step is repeated in a loop
System
3. Retrieves the topology
4. Create HwPC counters
5. Retrieves the metrics and counters
Sensor
6. Estimate processes consumption using system metrics using Scaphandre and Time ratio computations
7. Train the model if none is given and there is enough data
8. Use the model to estimate the process consumption
9. Check if the model is still accurate or retrain

Figure 5.1: Structure of the Scaphandre program

5.6. Summary

In the previous chapter, it was clear that none of the three programs we tested was a clear winner. We proposed that the best technique is to combine the three programs to create a new hybrid program that would take the best of each program.

In this chapter, we presented our new hybrid program named Geryon. As the base program, we chose Scaphandre, as it was the most versatile and easy to change. We then added the time ratio calculation from PowerJoular and the regression idea from SmartWatts. As the model used in the regression was missing some features that would help to make it more accurate, like using more data during training or using a more complex architecture, we added a way to train and use an external model.

In the next chapter, we will experiment with this new Geryon program and answer the last research question.

6 Analysis

Now that we have mixed the ideas from the three programs into a new hybrid program, we can start experimenting with it and analyze the results. We have separated the added functionalities into two experiments: the first one for the ideas we retrieved from the other programs, the time ratio and the regression, and the second one for the external model we added. For those experiments, we use one test and one machine: the Stress test on the Asus machine. This test was one of the most interesting tests we did in the experiment chapter, mainly because the Stress test includes many different workloads, and the Asus machine, with its high performance and consumption, allows us to easily see the difference between the estimations we made and the values we measured with the PDU.

6.1. Time ratio and regression

The first experiment we did was on the new time ratio and regression estimations we added to the hybrid program. Looking at the graphs in Figure 6.1, there are already some observations we can make. First, compared with the results in Figure 4.5, we still have spikes and missing values from the regression. This is, of course, logical due to how it works. However, the first interesting thing is that the time ratio is much more stable during the AIOL and the HDD workloads than PowerJoular was. This could be caused by the fact that we used a larger interval than the fixed one of PowerJoular of 1 second. Another interesting thing is that while the estimation of our time ratio and our regression follow a similar trend in the same way SmartWatts and PowerJoular did, our estimations are always higher than the measured values. The only exception is during the network workloads for the time ratio, where its estimations are accurate and close to the measured values. This observation can also be seen in Figure 6.2, where we have the two boxes completely above the 0W line, with both their IQR and median being higher than 100 watts. The only values that are below the 0W lines are the outliers. So even if we have a smaller variation than SmartWatts and PowerJoular, and thus smaller boxes than they have in Figure 4.5, we are entirely off with our estimations compared to their boxes centered around the 0W line. And this results in poor metrics, as shown in Table 6.1. For the regression, MAE and RMSE are more than three times higher than the ones we had with SmartWatts, with an R^2 that is three times lower. The comparison with the results between time ratio and PowerJoular is not as bad, but that is because PowerJoular poorly estimated the power consumption during that experiment.

After more analysis of the graphs, we found that the difference between the estimations and the reference value was constant over the complete run. So, we decided to compute a shifted version, where we subtracted the median from the values to bring it closer to the 0W line and see if it would improve the results. The results are promising, as all the estimations are closer to the

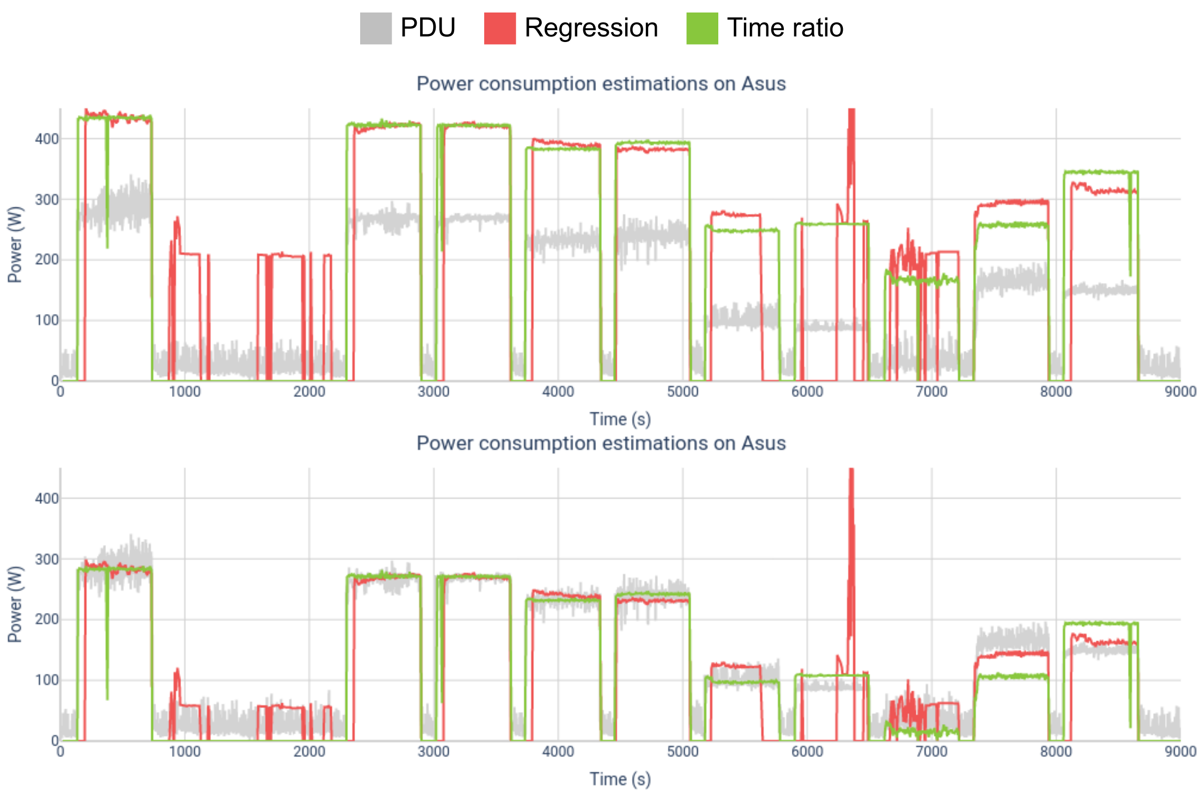


Figure 6.1: Estimations for the Stress test (without or with shift).

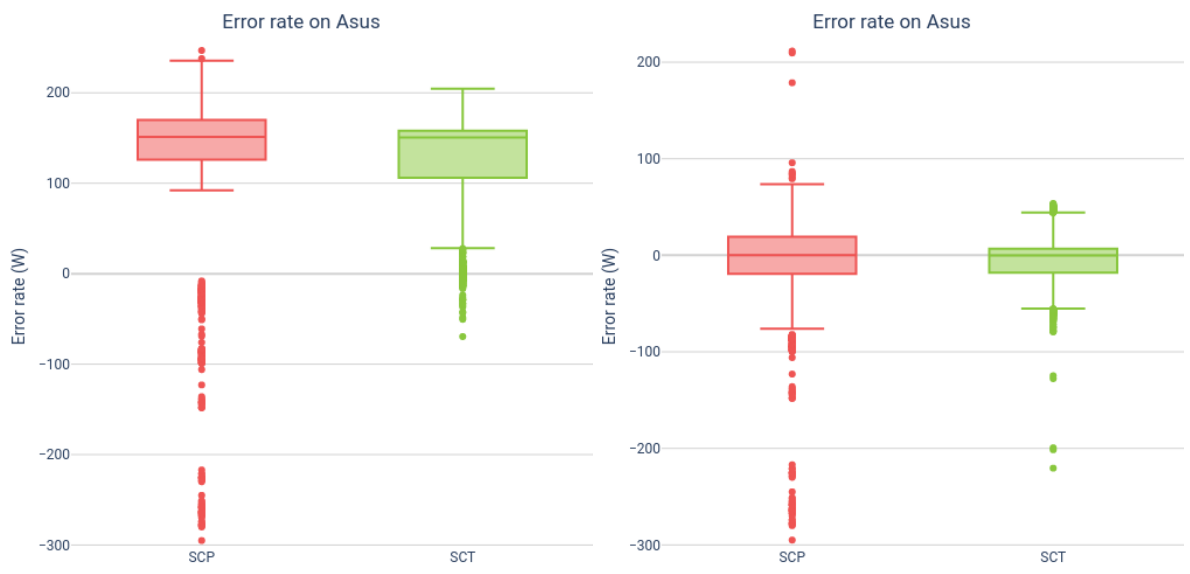


Figure 6.2: Error rate of the estimations on the Stress test (without or with shift).

	Regression			Time ratio		
Machine	MAE	RMSE	R^2	MAE	RMSE	R^2
Asus	144.450	154.895	0.290	127.467	139.990	0.420
Asus (with shift)	34.553	63.100	0.882	19.063	27.987	0.977

Table 6.1: Metrics of the estimations on the Stresstest test.

measured values. The box plots of both estimations are now centered and are even smaller than they were. They also are much smaller than the ones we had with SmartWatts and PowerJoular. The IQR of the regression is equal to 38.23, while the IQR of the time ratio is equal to 24.90. So, their estimations are more stable than the ones we had with SmartWatts and PowerJoular, where the IQR was equal to 85.23 and 131.75, respectively. In terms of metrics, the results are also much better. While the RMSE and the R^2 for the regression are still worse than the ones we had with SmartWatts, they are much closer to them than without the shift. The MAE is even better than the one we had with SmartWatts. For the time ratio, the results are much better than the ones we had with PowerJoular, which is logical as they were bad, but they are also even better than the ones we had with SmartWatts.

6.2. External model

Our second experiment was on the external model we added to the Geryon program. Before training the model, we had to collect the data. To do this, we did a first run of the experiment on the Asus machine, where we used our newly added collect exporter to write all the metrics into the CSV files. Then, we could train the model on the data we collected. To be able to show its performance in the most standalone way, we decided to train the model only on the HwPC counters, processes CPU usages, and system CPU frequencies, putting aside the default Scaphandre estimations and the time ratio estimations, even if they were strongly correlated with the power consumption.

When the model was trained, we computed some metrics to evaluate its performance before plugging it into the hybrid program. We did two different evaluations, and the metrics are reported in Table 6.2. The first evaluation used the test set we made during the data split just before training the model. This evaluation resulted in a mean absolute error of 5.5 watts, and the R^2 is close to 1. So, our model can predict the power consumption of the machine with a very high accuracy. However, it is important to remember that the data used for the training set and the test set are the aggregated values of the metrics we collected for each process at each timestep. Thus, with this set, the model predicts the power consumption of the whole system. But the final goal of the model is not to predict the power consumption of the machine, but to predict it for each process individually. So, we did a second evaluation where we used the original data from the processes to predict their power consumption. Then, we summed the estimations at each step to compare the result with the PDU values. The results are better than what we had with the regression and are even closer to the ones we had with SmartWatts, with a difference of less than 1 between their RMSE and 0.006 between their R^2. Like the regression, its MAE is also lower than the one we had with SmartWatts.

	Test set			Original data		
Machine	MAE	RMSE	R^2	MAE	RMSE	R^2
Headnode	5.500	7.451	0.9943	34.212	49.565	0.924

Table 6.2: Metrics during the training of the external model.

While those results are good, they are still only from the data we retrieved. To evaluate the external model more realistically, we need to plug it into Scaphandre and run a new Stress test. This is what we did for the last experiment. The estimations of the plugged model can be seen in Figure 6.3. Most of the time, the estimations are very close to the measured values, and most importantly, they are more stable than the ones we had with SmartWatts. Of course, we do not have

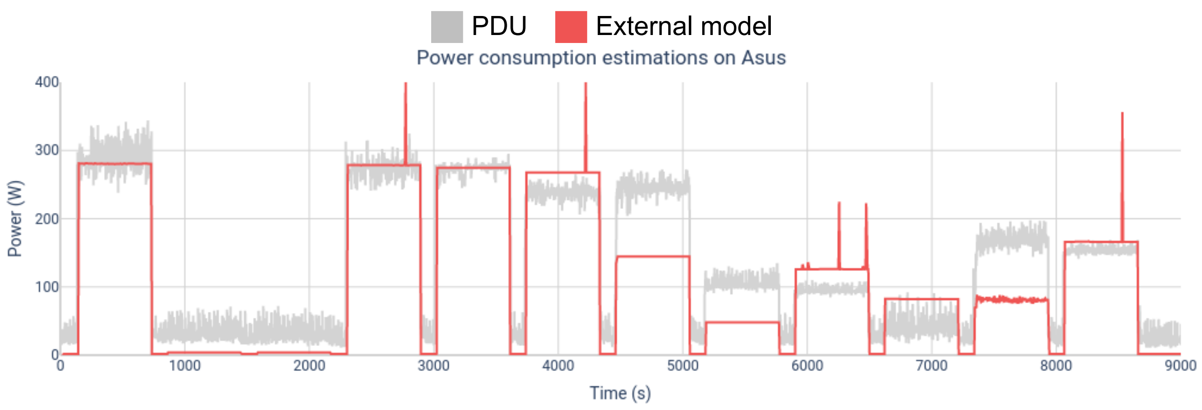


Figure 6.3: Estimations for the Stress test.

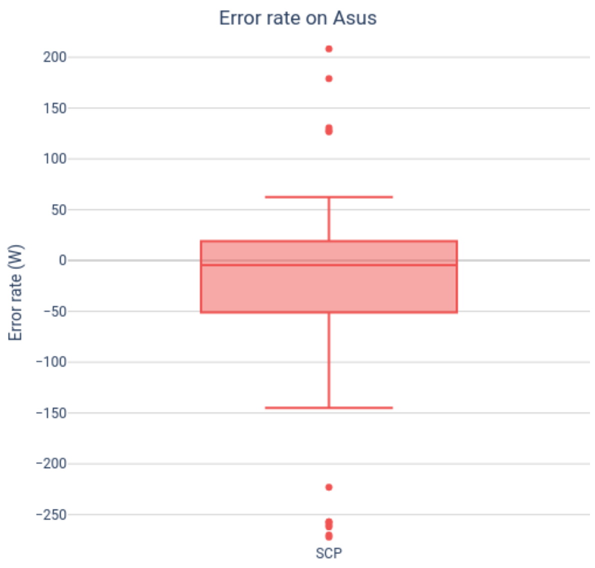


Figure 6.4: Error rate of the estimations on the Stress test.

missing values anymore, as we do not need to wait for the model to train. However, we still have some spikes. One of the possible reasons is that it could be caused by some unstable read of the HwPC counters. Looking at the box plot in Figure 6.4, we can see that the box is small. Its IQR and range are smaller than the one we had with SmartWatts. The IQR of the external model is equal to 68.22, while the one of SmartWatts is equal to 85.23. This observation shows that the estimations from the external model are more stable. But, while the box of SmartWatts was centered around the 0W line, the one of the external model is a bit lower but still keeps the line in its IQR, and its median of -4.441 is closer to the line than the one of SmartWatts with its -23. The metrics are good, but we can see that the MAE and RMSE are higher than those we had with the evaluation using the original data during training, although the increase is not that big, and the R^2 has improved a little.

	SmartWatts		
Machine	MAE	RMSE	R^2
Headnode	38.399	51.956	0.925

Table 6.3: Metrics of the estimations on the Stresstest test.

6.3. Discussion

The results of both experiments are interesting. For the first one, we can see that the time ratio and regression we added to the Geryon program are accurate. Both estimations are always higher than the measured values by a constant gap, but a simple shift using the median of the values can solve this problem. With this shift, the time ratio gives us much more stable estimations than PowerJoular, and the regression, while still having some spikes and missing values, is close to the measured values. The shift was surprising: our implementation follows the implementation of the SmartWatts and PowerJoular codes, but we did not need to do a shift when we ran experiments with those programs. Something that makes it even more intriguing is that the same gap is present in both implementations, meaning that it is not caused by a mistake in one of the codes. The only thing that they have in common is the RAPL value. So that could be the source of the gap. However, this shift is not an issue; It could be part of a calibration step that should be done once for each machine.

For the second experiment, the external model we added to the Geryon program also performed well, as, even with some spikes, it was more stable than SmartWatts. With those results, we can try to answer our last research question.

Can we design and implement a software-based power meter that would enhance the existing tools and overcome their limitations?

The answer is yes. First, we combined all three ideas from the programs into a single hybrid program, Geryon, without losing too much performance and accuracy. As we used the Scaphandre program as a base, we even improved the usage of the idea from both SmartWatts and PowerJoular. Now, the in-fly regression idea from SmartWatts is much easier to use as we no longer require two different binaries to run, and we do not require putting the monitored program in a CGroup. This also means no changes are needed to the system to activate the old CGroup version 1. For the time ratio idea from PowerJoular, implementing it into the Scaphandre programs allows it to take advantage of the other functionalities of the program, like choosing the interval of the measurements or exporting it in different ways. Having all three ideas together allows us to have a better understanding of the power consumption of the machine. While the regression is waiting for samples to be able to predict the power consumption, the time ratio and the Scaphandre computation already give us an estimation of the power consumption. And if one of the three is not working correctly, we still have the two others to rely on.

The external model we added to the Geryon program helped us to go even one step further. While our simple model is not as accurate as the estimations from SmartWatts, we are still close enough and more stable, with its IQR of 68.22 being lower than the IQR of SmartWatts of 85.23.

The models also allowed us to directly use the PDU values to train the model, which is a significant advantage as it has a more global view of the power consumption of the machine and not only the power consumption of the CPU and its embedded devices. Having an external model also has many advantages. First, we are not limited to the amount of data we want to use for the training. We could easily use the collect exporter on different workloads for a longer time to get a more global model. Secondly, we can easily change the architecture of the model. We could use a more complex model, like a neural network, to try and get better results, as we no longer have a limited amount of data and time to train the model. Finally, we can easily change the metrics we want in our training data. We could even add new measurements that are not directly correlated with the RAPL values but are correlated with the overall power consumption of the machine we get from the PDU. This could allow us to have a more accurate model, considering the power used by devices that are not directly linked to the CPU. The simple fact that we do not need the RAPL values allows us to use this external model idea on any machine, even if it does not have an Intel or an AMD CPU with the RAPL sensor. It only needs to have a PDU to measure the power consumption of the machine during the training of the model and a small script to train the model on the data we collected.

7 Conclusion

Since the creation of the ENIAC, the computers have evolved significantly. Their usage has also changed from being a tool for scientists and engineers to a device used by everyone and everywhere. They have become an essential part of our daily lives, but they have also become one of the most significant sources of energy consumption. The massive increase in their usage has made their energy consumption a significant concern both for the environment and the economy. However, while hardware manufacturers have made considerable progress in reducing the energy consumption of the different components of the computer, making them more energy-efficient, the same effort should be made at the software level. Software developers should be aware of the energy consumption of their applications and take it into account when designing and developing their softwares. However, there are not many tools available to help them with this task, and most of the existing tools are not generic. They are also limited to specific platforms or programming languages.

The primary goal of this thesis was to answer the main research question we stated in Section 1.2: *how to design and implement a generic software-based power meter that can accurately estimate the energy consumption of an application running on a computer?*. The generic term in the question means that the tool should be able to compute the power consumption on any machine, regardless of the hardware used. As the scope of the question was broad, we decided in Section 1.3 to focus only on the Linux operating system, as it is one of the most used on servers, workstations, and embedded systems. To achieve the goal, we divided it into three research questions.

RQ1 What are the existing software-based power meters, and how do they work and compare in terms of accuracy, performance, and features?

In Chapter 2, we started by presenting the three categories of methods to measure the power consumption of a system: using an external power meter, which has the highest accuracy but requires additional hardware and physical access to the system; using the embedded sensors that are available on some components like the CPU with the RAPL interface, but they are not available on all components; or using power models, which are the least accurate and relies on information provided with the components to make an estimation. All those methods can only estimate the power consumption at a system or component level. Still, they can also be used to compute the power consumption at a process level, using the resource usage of the process or its hardware events.

Then, we reviewed a list of the most popular tools and methods for estimating the power consumption of software applications, and we listed their features and limitations. Among all the tools we have listed, some were only interfaces to access the metrics from the embedded sensors easily, and only three of them were able to estimate the power consumption at an application or process level. We evaluate those three tools during various experiments on different machines to compare each tool's accuracy and performance using a PDU as the reference. We found that the tools were accurate, but their performance performance could vary a lot depending on the workload and the machine used. None of the three tools was a clear winner and was the best in all situations.

RQ2 What are the best techniques and metrics to use to estimate the energy consumption of an application?

After evaluating the three existing tools in our experiments, we could not find a clear winner, and we found that each tool had its strengths and weaknesses. All three tools use different techniques and metrics to estimate the power consumption of a process. One metric used by all three tools is the RAPL values provided by the CPU. This dependence means the tools can only run on machines that have a CPU with the RAPL interface, which is not the case for all machines.

PowerJoular and Scaphandre have a similar approach. They both use the CPU usage from the process to calculate the ratio from the total power consumption of the system that is due to the process. They then multiply this ratio by the RAPL values to estimate the power consumption of the process. They only differ in the way they calculate CPU usage. PowerJoular will compute the CPU usage on its own by using the total time of the process and the times the CPU was running or busy. Scaphandre, on the other hand, will directly use the CPU usage provided by the system. SmartWatts, which comes with PowerAPI, uses a different approach. It uses the hardware events provided by the CPU (HwPC) and trains a regression model on the fly that will estimate the power consumption of the process.

RQ3 Can we design and implement a software-based power meter that would enhance the existing tools and overcome their limitations?

As they were all good in their estimations, but we could not find a clear winner among the three tools, we decided to merge all of their techniques into one hybrid tool, Geryon. In Chapter 5, we presented the design of our tool. We used the Scaphandre program as the base of our tool as it is the easiest to use with only one binary to run. It is written in a well-structured code using Rust, and it already includes valuable features that allow us to change the interval of the measurements or the output format. The tests of those implementations showed us that while the regression was close to the accuracy of the original SmartWatts tool, it was a bit less accurate. However, the time ratio idea from PowerJoular was more precise and much more stable than its original implementation.

We also wanted to make it easier to train a new model on more data or to use more complex architecture. So, we implemented a new feature that allows us to easily store all the needed metrics from the processes and the system into CSV files and to plug an external model into the program quickly. We then trained a simple linear regression model on our collected data, using the PDU values as the target. The results were good as the metrics were close to those from the regression, with a better R^2 score but a slightly higher RMSE and MAE. It was also more stable than the regression, as it did need to wait for more data to make its predictions. Even if the results were not directly the best, we think that this new feature could be easily improved by using more complex models or by using more diverse data, like hardware counters that would not be directly correlated with the RAPL values, as they only report the consumption of the CPU and its embedded devices, but could still be used to estimate of the overall power consumption. As we also show, it no longer requires RAPL values to work, as it can be used with any external devices or tools that can provide the power consumption of the system.

So, to answer the main research question, with the design and implementation of the Geryon tool, we have shown that it is possible to design and implement a generic software-based power meter that can accurately estimate the energy consumption of an application running on a computer. Our tool is generic as it can be used on any machine, regardless of the hardware used, through the external model feature. It is also accurate and more stable, as it combines the best techniques from the existing tools. If one of the techniques is not correct or stable enough at a given time, the others can still provide a good estimation. As it removes the limits of time and data, the external model feature also allows us to use more complex architecture for the model and train it on a more diverse set of data, which would improve the accuracy of the tool. All of this makes our tool a good candidate for a good software-based power meter that can be used by software developers to estimate the energy consumption of their applications.

7.1. Future Work

While we have shown that our tool is a good candidate for a generic software-based power meter, some improvements could still be made to make its estimations even more accurate. One of the first improvements that could be made is to create and train more complex models. We have only trained a simple linear regression model, but more complex architecture, like neural networks, could be used to make more accurate predictions. The model could also be trained on a more extensive dataset generated by running more diverse workloads on the machine during a more extended period. This training would allow the model to be more robust and to make better predictions in a broader range of workloads. The model could also be trained on other metrics related to the process and the power consumption of the system, like hardware counters from other system devices that were not taken into account before. Those new metrics could also be about the overall state of the system, like the temperature of the CPU or the current fan speed, as they could also have an impact on the power consumption of the system. Other than improving the external model, we could also search for a better way to combine the estimations from the different techniques. Looking at the results, a simple average would not be the best way to combine them, but investigating deeper into the results could help to find the different cases where one technique is better than the others and find a way to determine when to put more weight on one method. Or perhaps the best way to combine the Scaphandre estimations with the time ratio and the regression would be to use them as features for the external model, combine them with other metrics, and let the model decide which one to use.

References

[1] Lorna Christie Aimee Ross. *POSTnote 677: Energy consumption in ICT*. Sept. 1, 2022. URL: https://post.parliament.uk/research-briefings/post-pn-0677/ (visited on 04/15/2024).

[2] AMD. “Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processors”. In: (Apr. 2017).

[3] Apple. *Energy Efficiency Guide for Mac Apps*. URL: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/PrioritizeWorkAtTheTaskLevel.html#//apple_ref/doc/uid/TP40013929-CH35-SW10 (visited on 02/27/2024).

[4] Yehia Arafa et al. “Verified Instruction-Level Energy Consumption Measurement for NVIDIA GPUs”. In: *CoRR* abs/2002.07795 (2020). arXiv: 2002.07795. URL: https://arxiv.org/abs/2002.07795.

[5] David H. Bailey. *NAS Parallel Benchmarks*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1254–1259. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_133. URL: https://doi.org/10.1007/978-0-387-09766-4_133.

[6] François Chollet et al. *Keras*. https://keras.io. 2015.

[7] Cisco. *Cisco Annual Internet Report (2018-2023) White Paper*. Mar. 9, 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (visited on 04/15/2024).

[8] Sustainable Computing. *Kubernetes Efficient Power Level Exporter (Kepler)*. URL: https://sustainable-computing.io/ (visited on 02/27/2024).

[9] Benoit Courty et al. *mlco2/codecarbon: CodeCarbon*. Aug. 2023. DOI: 10.5281/zenodo.4658424. URL: https://doi.org/10.5281/zenodo.4658424.

[10] We Are Social DataReportal Meltwater. *Number of internet and social media users worldwide as of January 2024 (in billions) [Graph]*. Jan. 31, 2024. URL: https://www-statista-com.tudelft.idm.oclc.org/statistics/617136/digital-population-worldwide/ (visited on 04/15/2024).

[11] ESnet. *iperf3 repository*. URL: https://github.com/esnet/iperf (visited on 02/27/2024).

[12] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. “SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers”. In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 2020, pp. 479–488. DOI: 10.1109/CCGrid49817.2020.00-45.

[13] Daniel Hackenberg et al. “An Energy Efficiency Feature Survey of the Intel Haswell Processor”. In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 2015, pp. 896–904. DOI: 10.1109/IPDPSW.2015.70.

[14] David Harrington, Bert Wijnen, and Randy Presuhn. *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*. RFC 3411. Dec. 2002. DOI: 10.17487/RFC3411. URL: https://www.rfc-editor.org/info/rfc3411.

[15] IBM. *Perf Wiki*. URL: https://perf.wiki.kernel.org/ (visited on 02/27/2024).

[16] Intel. *Intel Performance Counter Monitor repository*. URL: https://github.com/intel/pcm (visited on 02/27/2024).

[17] Mathilde Jay et al. “An experimental comparison of software-based power meters: focus on CPU and GPU”. In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2023, pp. 106–118. DOI: 10.1109/CCGrid57682.2023.00020.

[18] Kashif Khan et al. “RAPL in Action: Experiences in Using RAPL for Power Measurements”. In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3 (Jan. 2018). DOI: 10.1145/3177754.

[19] Colin Ian King. *powerstat repository*. URL: https://github.com/ColinIanKing/stress-ng (visited on 02/27/2024).

[20] Colin Ian King. *stress-ng repository*. URL: https://github.com/ColinIanKing/powerstat (visited on 02/27/2024).

[21] Klaus-Dieter Lange et al. “SPECpower_ssj2008: driving server energy efficiency”. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE ’12. Boston, Massachusetts, USA: Association for Computing Machinery, 2012, pp. 253–254. ISBN: 9781450312028. DOI: 10.1145/2188286.2188329. URL: https://doi.org/10.1145/2188286.2188329.

[22] James H. Laros, Phil Pokorny, and David DeBonis. “PowerInsight - A commodity power measurement capability”. In: *2013 International Green Computing Conference Proceedings*. 2013, pp. 1–6. DOI: 10.1109/IGCC.2013.6604485.

[23] linux.die.net. *snmpget(1) - Linux man page*. URL: https://linux.die.net/man/1/snmpget (visited on 02/27/2024).

[24] Jens Malmodin et al. “ICT Sector Electricity Consumption and Greenhouse Gas Emissions - 2020 Outcome”. In: *SSRN Electronic Journal* (2023). DOI: 10.2139/ssrn.4424264.

[25] Corey Malone, Mohamed Zahran, and Ramesh Karri. “Are hardware performance counters a cost effective way for integrity checking of programs”. In: *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*. STC ’11. Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 71–76. ISBN: 9781450310017. DOI: 10.1145/2046582.2046596. URL: https://doi.org/10.1145/2046582.2046596.

[26] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[27] Adel Noureddine. “PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools”. In: *18th International Conference on Intelligent Environments (IE2022)*. Biarritz, France, June 2022.

[28] NVIDIA. *NVIDIA Management Library (NVML)*. URL: https://developer.nvidia.com/management-library-nvml (visited on 02/27/2024).

[29] NVIDIA. *System Management Interface SMI*. URL: https://developer.nvidia.com/nvidia-system-management-interface (visited on 02/27/2024).

[30] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[31] B. Petit. *Scaphandre documentation*. URL: https://hubblo-org.github.io/scaphandre-documentation/ (visited on 02/27/2024).

[32] Satyabrata Sen, Neena Imam, and Chung-Hsing Hsu. “Quality Assessment of GPU Power Profiling Mechanisms”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 702–711. DOI: 10.1109/IPDPSW.2018.00113.

[33] I. SPIRALS. *PowerAPI*. URL: https://powerapi.org/ (visited on 02/27/2024).

[34] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[35] Patrick Christian Konsor Timothy McKay. *Intel Power Gadget*. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html (visited on 02/27/2024).

[36] Jan Treibig, Georg Hager, and Gerhard Wellein. “LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments”. In: *2010 39th International Conference on Parallel Processing Workshops*. 2010, pp. 207–216. DOI: 10.1109/ICPPW.2010.38.

[37] Arjan van de Ven. *PowerTOP repository*. URL: `https://github.com/fenrus75/powertop` (visited on 02/27/2024).

[38] Brendan Reidenbach Vida Rozite Emi Bertoli. *Data Centres and Data Transmission Networks*. July 11, 2023. URL: `https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks` (visited on 04/15/2024).

[39] Vincent M. Weaver et al. "Measuring Energy and Power with PAPI". In: *2012 41st International Conference on Parallel Processing Workshops*. 2012, pp. 262–268. DOI: `10.1109/ICPPW.2012.39`.

[40] Hui Zou and Trevor Hastie. "Regularization and Variable Selection Via the Elastic Net". In: *Journal of the Royal Statistical Society Series B: Statistical Methodology* 67.2 (Mar. 2005), pp. 301–320. ISSN: 1369-7412. DOI: `10.1111/j.1467-9868.2005.00503.x`. URL: `https://doi.org/10.1111/j.1467-9868.2005.00503.x`.