On the Energy Impact of Programming Frameworks on Software Services

Mohammed Chakib **BELGAID**

Supervisors: Pr. *Romain* ROUVOY,
Pr. *Lionel* SEINTURIER

University of Lille

This dissertation is submitted for the degree of
Doctor of Philosophy

École Doctorale MADIS September 2022

I would like to dedicate this thesis to my loving parents ...

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Mohammed Chakib BELGAID
September 2022

Acknowledgements

And I would like to acknowledge ...

Abstract

This is where you write your abstract …

Contents

List of Figures

List of Tables

Chapter 1

Introduction *TODO : missing*

Nowadays, computers are invading our daily lives, from work to leisure, from fancy smartphones to an embedded peacemaker that regulates the heartbeat of people. As human beings, we are known to use tools to enhance our bodies. And thanks to computers, we pushed that step even further, to the point where now we are using machines to extend our brains. Some even argue that one day they will replace us, and therefore we have created our own ending.

from 4.3 exajoules in 2018 to 5.8 exajoules in 2025[1]. https://www.iea.org/reports/data-centres-and-data-transmission-networks

The Internet of Things (IoT) is one of the most popular topics in computer science and engineering, and it is expected to be a very important part of our lives in the future However, the energy consumption of ICT equipment is also expected to increase,

Well, this is the problem for the future generations. For the moment, the main concern of humanity is to keep this planet liveable until we find another alternative. The number of data centers is expected to increase from 1.6 million in 2018 to 2.1 million in 2025, according to [16].

The number of people connected to the Internet has increased by 4.4 billion in 2019, reaching 4.54 billion worldwide, or 59.2% of the world population, according to the Internet World Stats[2].

All these new activities have increased the overall environmental footprint of the Information and Communication Technology (ICT) sector, which is estimated to be responsible for approximately 4% of the greenhouse gaz (GHG) emissions worldwide in 2020 with a worrying 8% growth rate, according to the French think tank The Shift Project [137], or 2% according to [13], a similar number to the aviation sector contribution.

[137]: https://www.theshiftproject.org/article/ict-environmental-impact/

[1] https://www.statista.com/statistics/271139/energy-consumption-of-ict-worldwide
[2] https://www.internetworldstats.com/stats.htm

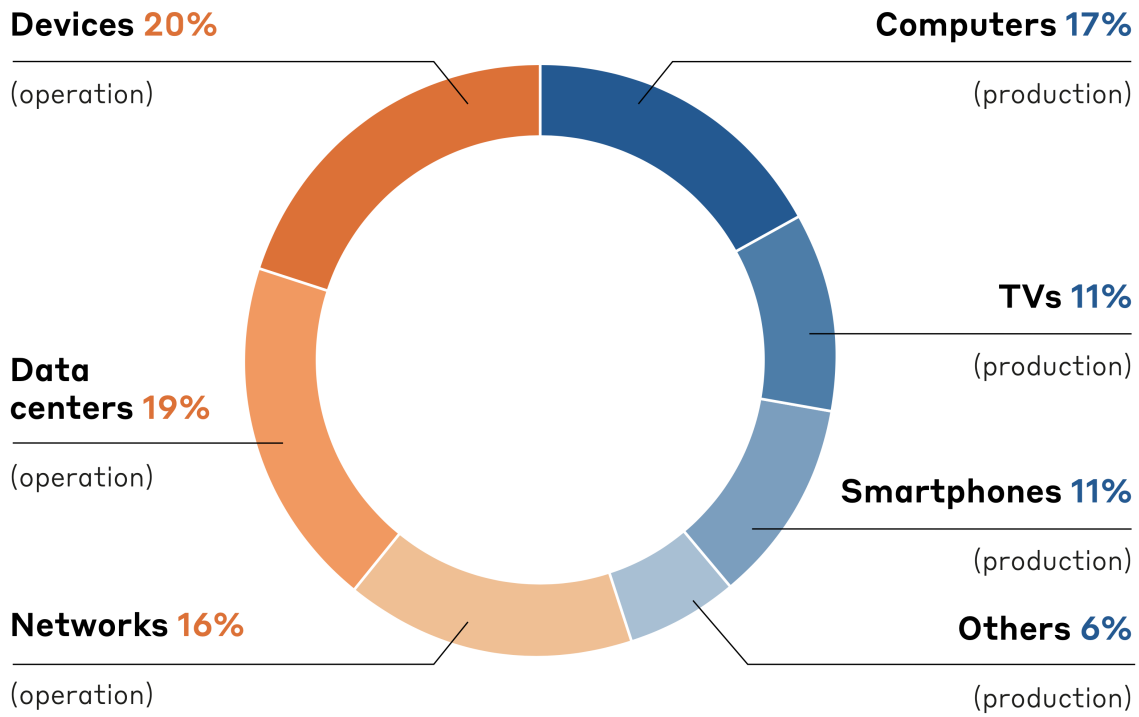


Figure 1.1: Final energy consumption of digital technologies by item in 2017

[13]: https://www.theguardian.com/environment/2020/jan/06/tech-industry-emissions-soar-at-double-rate-of-aviation-and-shipping

Unfortunately those machines doesn't help that much. as according to ....

Researchers are trying to solve this issue through different angles. While some scientists are trying to find an alternative green source to generate energy. others focus on reducing this energy consumption. In computer science we are concerned with the ladder solution Therefore most of the works are done on tunning the software and the hardware in order to have a more efficient way use this energy. In this thesis we are trying to find a way to make the energy consumption of the computer to be as low as possible. Our approach is to reduce the energy consumption of the software services by changing certain parameters, such as platform, programming language, and/or the design pattern.

The best way to do so is to formulate a theory behind the energy consumption of algorithm, such as the complexity and the o notation. Unfortunately this is not possible in the current state of the art. due to the lack of knowledge about the energy consumption of the algorithms, and the strong correlation between this consumption and the hardware configuration.

Unlike algorithm optimization in the field of performance, which is agnostic toward the platform, the energy consumption of the algorithms is dependent on execution environment.

Therefore, for the moment we will start by formulating some hypothesis and explore them using empirical analysis.

Our work will be presented through the following chapters:

1. 1.3: Where we discuss the work done on the energy consumption and optimization in software engineering
2. 3: It will present a set of guidelines and tools to help practitioners measure the energy consumption of their algorithms.
3. : we will present the impact of programming languages on the energy consumption of the algorithms.
4. : it will discuss the behaviour of python and the possibile ways to tune it in ordrer to reduce the energy consumpion
5. : will present a study on java programming language and the impact of the JVM choice on the energy consumpion
6. : as a prespective we introduce the impact of parallelisme on the enrgy consumption on time agnostic cases

....

1.1 Motivation

....

1.2 Research Contributions

1. Introduces
2. Shows how ....
3. Proposes ...

1.3 Publications

1. ...

2. ...

3. ...

Chapter 2

State of the Art

2.1 Introduction

Efficiency in energy usage is a well-known topic. In the majority of fields, the purpose is to minimize the energy consumption of electrical devices and components. Modern times even see energy classification (A, B…F) for homes, cars, and electronic products, to provide the consumer an indication of the energy consumption of their devices, which will reflect on their power bill.

This criterion is extended even to the hardware components of a computer. Figure 2.1 compares Intel CPUs i9-12900KS and i9-12900KF.[1] The difference between these two CPUs is that the KS has an unlocked multiplier, allowing it to be overclocked. As a result, the basic consumption is less than the KF. This statistic also estimates the average power consumption of these two CPUs each day, as well as the monetary equivalent, to make people more aware of the values of energy consumption rather than the raw data.

In computer science, the objective is essentially the same. Numerous studies have been conducted on energy optimization. Some of these studies concentrate on minimizing energy consumption at the hardware level, while others optimize energy consumption via software.

As an example, Avgerinou et al. evaluated the development of power use effectiveness (PUE) in data centers that belongs to various organizations participating in the European code of conduct for energy efficiency program [6]. The research found a gradual decline in the PUE of data centers, which measures the ratio of the overall energy supplied to the energy used by IT equipment. A low PUE implies that the majority of energy is utilized to power the data center's IT equipment, while just a small amount is needed for cooling and lighting.

[1] https://www.cpubenchmark.net/compare/Intel-i9-12900KS-vs-Intel-i9-12900KF/4813vs4611

	Intel Core i9-12900KS	Intel Core i9-12900KF
Max TDP	150W	241W
Power consumption per day (kWh)	0.3	0.5
Running cost per day	$0.075	$0.120
Power consumption per year (kWh)	109.5	175.9
Running cost per year	$27.38	$43.98

Shown CPU power usage is based on linear interpolation of Max TDP (i.e. max load). Actual CPU power profile may vary.

Figure 2.1: Electrical cost comparison between two CPUs.

We will place a greater emphasis on the software level to decrease the amount of energy that is used, more particularly in the execution phase of the program cycle. We will be proceeding through an empirical analysis of the energy consumption of the software while changing some components of the source code without impacting its behavior. To do this, we will elaborate on a benchmarking process and a set of tools that are intended to assist practitioners in better comprehending and optimizing the energy usage of their applications. Thus, we will begin by examining the state of the art in empirical analysis and retrieving the best empirical experimentation methodologies in the research field. Then, we will narrow these practices down to computer science so that we can finally adapt them to energy consumption.

Section 2.2 will discuss the pitfalls and best practices associated with empirical research before applying them to our field of interest. After that Section 2.3 describes software energy measurements. It provides examples of hardware and software measuring instruments and describes their differences, benefits, and drawbacks. It also examines the sources of energy measurement variations, which represent a significant obstacle to achieving precise readings and higher accuracy. Then, in Section 2.5, we will go through some of the previous work on improving the energy consumption of software.

2.2 Benchmarking

This section will go through the flaws and best practices of empirical research before applying them to our topic of study.

2.2.1 Threats & Challenges

A successful benchmark must meet three criteria. First, it must be **reproducible** for others to imitate it. Second, the findings should be **accurate**, which implies that we should expect the same results each time we run the benchmark. Finally, it should **represent** reality. In other words, the experiment's findings should be applicable outside of the research lab as well. The aim of **representativeness** in this thesis is the manufacturing environment. As a result, the experiments should reflect what happens in production contexts.

Reproducibility

Experiment reproducibility is frequently listed as one of the most difficult challenges faced by researchers. Reproducing an experiment has been one of the major tools science has used to help establish the validity and importance of scientific findings since the Philosophical Transactions of the Royal Society were established in 1665 [47]. Many of the outcomes are not reproducible,[2] which led to a *replication crisis*. As a result of the crisis affecting the majority of empirical studies, most reviews now include reproducibility as a minimum standard for judging scientific merit [90]. One of the criteria for supporting reproducibility is the publication of the dataset and the algorithms run on the raw data to derive the results. There is even some disagreement about what the terms "reproducibility" or "replicability" by themselves mean [41]. According to [31], *replicability* extends *reproducibility* with the ability to collect a new raw dataset comparable to the original one by re-executing the experiment under similar conditions, instead of just the ability to get the same results by running the statistical analyses on the original data set.

Accuracy

According to Oxford, *accuracy* means "technical The degree to which the result of a measurement, calculation, or specification conforms to the correct value or a standard". In our case, this means the ability to run the benchmark multiple times with low variation. This can be achieved by controlling the experiment environment, allowing less room for random factors.

[2] Trouble at the lab, The Economist, 19 October 2013; www.economist.com/news/briefing/ 21588057-scientists-think-science-self-correcting- alarming-degree-it-not-trouble.

In biology, chemistry, and electronics, they use clean rooms, which are environments where pollutants like dust, airborne microbes, and aerosol particles are filtered out and factors like humidity, airflow, and temperature can be regulated. As for empirical analysis, the accuracy can be measured by numeric metrics such as the variance, the standard deviation (STD), and the interval inter quartile (IRQ). Section 2.4 will go over the accuracy in the subject of energy optimization.

Representativeness

As obvious as it seems, the reason for executing benchmarks is to validate ideas so we can use them in real life. However, this means that those benchmarks have to represent reality somehow. In Social sciences, this can be achieved by selecting a representative sample size. Omair et al. presents a guideline on to achieve a such representativeness. As for computer science. The field of benchmarking is still in its infancy, and there is no consensus on how to achieve representativeness. howerver many attemps have been made to provide a set of benchmarks for specific purpose. For example, the Standard Performance Evaluation Corporation (SPEC) provides a set of benchmarks for CPU performance evaluation. this sets covers a wide range of use cases such as CPU17 for testing the CPU [3], SPECviewperf[4] for Graphic usage and one can cite StressNg when it comes to benchmark the hardware components, the SPEC benchmark when it comes to benchmark the software performance, and SPECPower[5]. NASA Parallel Benchmarks (NPB)[6] and HPCchallenge[7] are two other examples of benchmarking sets that are created to represent the high performance computing. As for programming languages we can cite Computer Language Benchmarks Game [8], which is a collection of benchmarks for various programming languages. The benchmarks are designed to be small, self-contained, and easy to implement in any language. The benchmarks are also made in a way to represent of most of the typical real-world workloads in an isolated manner. Dacapo [13] and renaissance [96] are other examples of benchmarking sets that are created to represent the Java Virtual Machine (JVM) performance. On the other hand, a new sort of test has arisen within the software development life cycle. This type is known as stress testing, and it is used to assess the software's robustness and reliability before releasing it to

[3]https://www.spec.org/cpu2017/
[4]https://gwpg.spec.org/benchmarks/benchmark/specviewperf-2020-v3-1/
[5]https://www.spec.org/power_ssj2008
[6]https://www.nas.nasa.gov/publications/npb.html
[7]https://www.hpcchallenge.org/
[8]https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

the public. gatling [9] and TCPCopy[10] are great examples of stress testing tools that are used to test the performance of server applications.

Impact of these Challenges in the Empirical Research

In their work [105], Van-der-Kouwe *et al.* investigated 50 papers published in top venues to find out that Tier-1 papers commit an average of five benchmarking crimes. To analyze the magnitude of the phenomenon, they have identified a set of 22 "benchmarking crimes" that threaten the validity of the system.

2.2.2 Proposed Solutions

Researchers have proposed several solutions to overcome these challenges in the computer science field. We will discuss some of them below. First, we start with the work of Mytkowicz et al., where they evaluated 133 studies from ASPLOS, PACT, PLDI, and CGO, to find out that none of the experimental findings papers appropriately considered measurement bias. Which can lead derive incorrect results from an experiment if a seemingly insignificant feature of the experimental design is altered. They treated this problem by proposing two strategies for detecting measurement bias by using causal analysis and preventing it with setup randomization [80]. another study that was published in the book "Measuring computer performance: a practitioner's guide" [69],Lilja examined performance indicators and gave in-depth treatment of benchmark program tactics. they provided clear explanations of the basic statistical methods required for interpreting measured performance data. They also outlined the overall design of the experimental method and demonstrated how to collect the most information with the least amount of work. This practical book will appeal to anybody seeking a comprehensive, yet intuitive, grasp of computer system performance analysis.

Bukh wrote a book about computer performance analysis, where they discussed some familiar topics that are relevant to statistical analysis, such as null hypotheses, chi-squared tests, regression, discrete event simulation, Bayes' theorem, how and when to use them for experimental design, measurement, simulation, and modeling for computer systems. The article [17] provides an intellectual framework for understanding the pervasiveness of mistakes in the scientific discovery process and presents methodological, cultural, and system-level techniques for minimizing the frequency of often seen errors.

[9] https://gatling.io
[10] https://github.com/session-replay-tools/tcpcopy

Another article [89] expands on this line of thought by evaluating the uncertainty caused by replications in the new research. They offered some strategies to capture uncertainty in inferential investigations, such as cross-study validation and ensemble models.

In their paper [103], the authors found that, even though it's a big step in the right direction, journal policies that require authors to give back digital scholarly objects after publication, like the data and code that back up the claims, don't get more than half of these objects back. Then, using these artifacts, about a fourth of the published computational claims in the study could be made. They suggested putting out the claims in the literature and the digital scholarly objects that back them up at the same time.

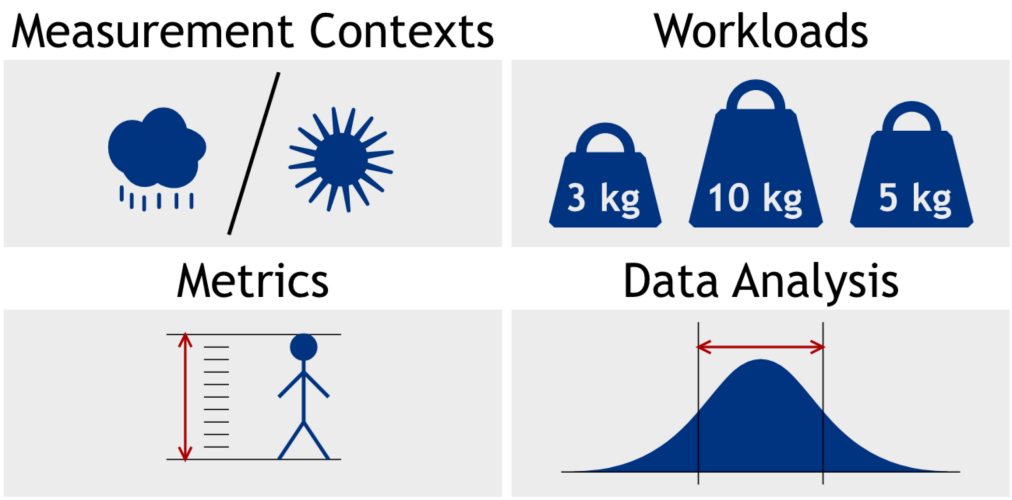


Figure 2.2: The proposed decomposition of an experiment [11]

Finally, to make unify the benchmarking methodology across different research works in the field of computer science, we can cite the paper [11]. In their approach, they divided any empirical experiment into four components. Figure 2.2 presents these components, which are:

1. *measurement contexts* that indicate the software and hardware components that will alter or remain constant during the experiment.

2. *workloads* which identify the benchmarks to use in the experiment, as well as their inputs;

3. *metrics* that specify the attributes to be measured and how to assess them.

4. *Data analysis* shows how to examine data and evaluate the outcomes of the analysis to offer insight into the assertions that arise from the study.

The work of this thesis will be based on this approach since it provides a unified methodology for benchmarking and evaluating the performance of different systems.

2.3 Energy Measurement

Now that we have discussed the importance of benchmarking and the different approaches that can be used to evaluate the performance of a system, we will focus on energy measurement, which will be the main metric in this thesis. Therefore, in this section, we will discuss the different approaches that can be used to measure the energy consumption of a system. Many studies have been conducted to estimate such energy consumption that varies from static analysis of the source code to infer its energy consumption like Pereira et al. where they provided a tool to highlight the most energy consuming parts of the code [91]. The key advantage of this approach is that it allows you to estimate a program's energy usage without executing it. Unlike program complexity, energy consumption is strongly dependent on the execution environment. As a result, static analysis may not accurately represent the behavior of the same program when run in a production context. To address the issue of representativeness, many researchers measure the energy consumption of programs as they run. As a result, we will get more accurate results. There are various tools for measuring energy, and they cover a wide range of applications depending on how accurate and precise the results must be on the one hand, and the price that practitioners are prepared to pay for such accuracy and precision on the other.

According to Hackenberg et al., there are four main criteria to evaluate an energy measurement tool [44]:

- *Spatial granularity: the more specific the target of monitoring we can measure, the more efficient we can do optimization since we will know what causes the pitfalls of the energy consumption,*
- Temporal granularity: same as spatial granularity, temporal granularity helps us to identify the sequence of code that need to be optimized,
- *Scalability: this is mainly related to the cost of the tools and the ease of their integration for our system,*
- Accuracy: to eliminate extra hazards and get a more representative measurement.

We believe that accuracy can be extracted from those criteria since it is a result of the combination of the two first ones. therefore, we will focus more on the first 3 criteria from later on. Below, we will discuss some of the well-known tools that are used in literature.

2.3.1 Hardware Tools

Nowadays, most high-performance computing systems (HPC) implement a tool to report the energy consumption of the nodes for the sake of monitoring and administration. Those tools are mainly integrated within the power supply units (PSU) or the power distribution units (PDU). Then, they provide an interface and a log to follow the history of energy consumption. Despite their scalability and ease of integration. Such tools lack both spatial and temporal granularity since they monitor the whole energy of the nodes, and most of the time they have a very low sampling frequency. Most of those tools are provided directly by the manufacturers. Such as *IBM EnergyScale technology [77, 22, Caldeira et al.] or* Dell poweredge [73], MEGware Cluststafe [16]. As said earlier, the true purpose of those tools is more monitoring than analyzing energy consumption. WattsUp Pro, is a device that can be installed between the power source of the machine and the system under test. It allows a sampling frequency up to 1 Hz and has an internal memory to store a wide variety of data, such as the maximum voltage, and current, that later can be exported via USB port for personal usage or lined to some graph programs like Logger Pro or LabQuest. The main advantage of this tool is the ability to monitor the energy consumption from a different device which will reduce the risk of interference with the energy consumption of the test [50]

Despite its high temporal granularity, WattsUp Pro lacks in term of spatial granularity since it monitors the energy consumption of the whole system. To have a finer granularity we need to isolate the energy consumption of each component.

PowerMon and its upgraded version powerMon2 [9] are based on a micro-controller chip that can monitor up to 6 channels (8 for powermon2), simultaneously. Therefore, we are able to monitor the power consumption of 4 devices at the same time. The frequency sample of this tool is up to 50 Hz, with an accuracy of 1.2%. Powermore2 comes in a smaller size that can fit within 3.5 inches rack drive.

PowerInsight [65] is another fine-grained measurement tool that is based on an ARM BeagleBone processor [26], which can measure up to 30 channels simultaneously with a frequency of 1KHz per channel.

powerpack [40] in the other hand is an API that synchronizes the data gathered from other monitoring tools such as Watt's Up Pro, NI and RadioShack pro and the lines of code.

Other monitor tools have been realized by the manufacturers, such as IBM Power executive [61], which allows their customers to monitor the power consumption and thermal behavior of the of BladeCenter systems in the data center.

Accoring to the work of Vasques et al. andWang et al. The CPU is the part responsible for the most energy consumption in a data center[110, 111]. Hence, the finer we go to measure this energy consumption, the better it is for our work. Fortunately, Intel and later AMD

proposed a tool that estimates the power consumption of different parts of the CPI based on counter performances. RAPL (*RUnning Average Power Limit*) [43, 45] is a set of registers that was introduced by Intel in their CPU since Sandy bridge generation, and later it was followed by AMD, since Family 17h Zen.

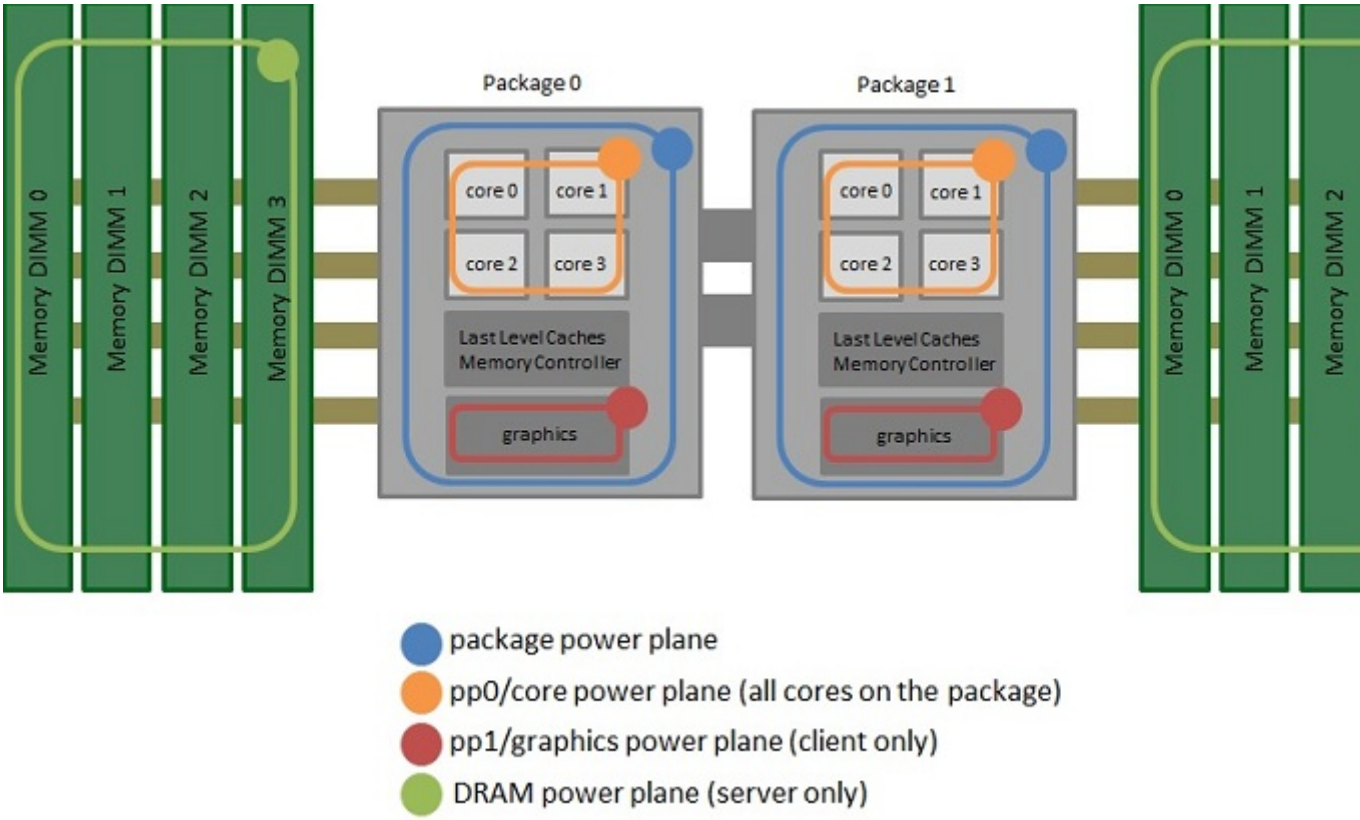


Figure 2.3: INTEL RAPL scopes

Figure 2.3 shows the different scopes that can be monitored using RAPL. The CPU package can be monitored for both server and desktop processors. However, DRAM is only available on server CPUs, whilst the integrated GPU is only available on desktop processors. The advantage of such an approach is the absence of any intrusive measurement tools. Furthermore, they have a high temporal granularity with a sampling frequency of up to 1 KHz [51].

With a similar approach, we can find NVIDIA reporting tools, such as GPU TESLA [20] and the NVML library [35].

2.3.2 Software Tools

Software-based measurement tools are based on other hardware tools to monitor energy use. Granularity is the core value of these technologies, unlike hardware tools, which only provide the total energy usage of the system/component (computer, server, motherboard, etc.) in most cases. Because they are frequently constructed on empirical estimations and data learning methods, they drop in accuracy.

Many software measuring tools understand the behavior of a power model and provide estimates of energy usage. This model is then used to allocate the observed energy consumption among different execution entities, such as processes, control groups, threads, or even code lines.

The first examples of software measurement tools are PowerAPI [27], SmartWatts formula [37] and SelfWatts [38]. These tools collect global energy consumption measurements from RAPL and use other system events such as cache misses/hits and CPU frequency evolution (DVFS) via a sensor to construct a power model of the control groups (system control groups, docker containers, kubernetes pods, etc.) using a Ridge regression. The model continuously learns and improves its precision of real-time energy usage data with a maximum frequency of 100 Hz. The instrument has a decentralized, lightweight design. Only the lightest sensors required for data collection and transmission are put into the monitored devices. The SmartWatts formula is then executed on the primary server to construct the model that permits assigning the energy usage for each operating control group. PowerAPI is only compatible with Linux on a bare-metal physical computer.
WattWatcher [66] is a multi-core power measuring framework that provides process-level energy measurements. This program uses power models to predict process energy usage. It uses CPU events that are passed from the measured node to a model generator node to construct the power model. It works by combining a description of the CPU with a list of the hardware events through multiple calibration phases to build a robust model.

Joulemeter [62, 54] is a Microsoft software that estimates the energy usage of Windows running applications down to the process level by using power models (for CPU, memory, and drives). It employs low-overhead power models to infer power consumption from resource utilization during runtime, and it provides power-limiting features for virtual machines. Previous Joulemeter tests [55] demonstrated that the instrument provides a less accurate estimation of energy use that differs greatly from the real one. To adjust its models to the hardware on which it operates, Joulemeter must first go through a calibration step. It can only monitor one process at a time with a frequency of 1 Hz.

JRAPL is another example of an energy measurement tool estimating tool that has been utilized in a variety of publications [42, 71]. This software enables the energy usage of

Java programs, functions, or even a block of code lines to be profiled and measured. The measurements are heavily reliant on the data supplied by RAPL. As a result, the global energy consumption collected by RAPL between two timestamps (the start and finish of the code to measure) is used to calculate the energy consumption of the Java code. Tests using jRAPL should be done on a well-configured machine to minimize the impact of the operating system and user operations on the overall energy consumption of jRAPL.

Another process-level energy usage measuring tool is Jolinar [53, 82]. The tool does not need a calibration phase and relies on pre-established power models based on hardware metrics (TDP, disk I/O rate, and so on). These settings must be identified and supplied by the user for his machine. Jolinar is only capable of measuring the energy consumption of one application at a time. At the end of the execution, the tool provides the CPU, DRAM, and disk energy usage of the main process. Jalen [83] is another tool that profiles and monitors the energy usage of a Java program. Unlike jRAPL, Jalen can cover the scope down to the function level. It gathers data using code instrumentation and statistical sampling at a predetermined pace. Because of the overhead that code instrumentation may incur, the authors recommend utilizing the second option. Every 10 ms, Jalen records the JVM's stack trace together with the CPU time of threads and computes statistics about method calls. These statistics are then utilized to calculate each method's energy usage.

2.4 Energy Variations

In theory, using an identical CPU, the same memory configuration, and similar storage and networking capabilities should increase the accuracy of physical measurements. Unfortunately, this is not possible when it comes to measuring the energy consumption of a system. Applying the benchmarking guidelines and repeating the same experiment within the same configuration is not sufficient to reproduce the same energy measurements, not only between identical machines, but even within the same machine. This difference—also called *energy variation* (EV)—has become a serious threat to the accuracy of experimental evaluations.

Figure 2.4 illustrates this variation problem as a violin plot of 20 executions of the benchmark *Conjugate Gradient* (CG) taken from the *NAS Parallel Benchmarks* (NBP) suite [7], on 4 nodes of an homogeneous cluster (the cluster Dahu described in Table 3.1) at 50 % workload. We can observe a large variation of the energy consumption, not only among homogeneous nodes but also at the scale of a single node, reaching up to 25 % in this example.

Some researchers started investigating the hardware impact of the energy variation of power consumption. As an example, one can cite [14, 104] who reported that the main

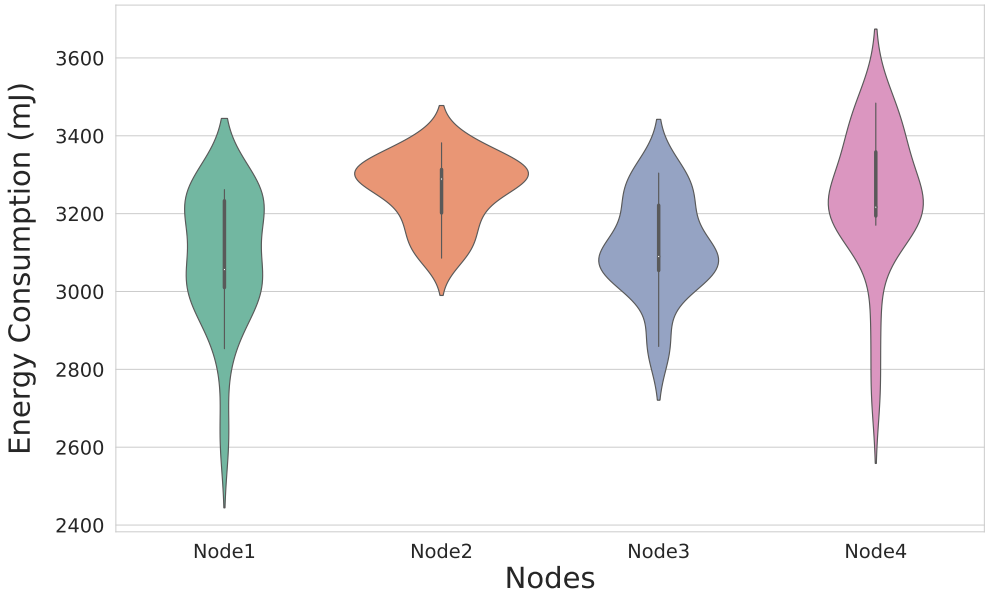


Figure 2.4: CPU energy variation for the benchmark CG

cause of the variation of the power consumption between different machines is due to the **CMOS** manufacturing process of transistors in a chip. [49] described this variation as a set of parameters, such as CPU Frequency and the thermal effect.

2.4.1 Studying Hardware Factors

This variation has often been related to the manufacturing process [25], but has also been a subject of many studies, considering several aspects that could impact and vary the energy consumption across executions and on different chips. On the one hand, the correlation between processor temperature and energy consumption was one of the most explored paths. Kistowski *et al.* showed in [56] that identical processors can exhibit significant energy consumption variation with no close correlation with the processor temperature and performance. On the other hand, the authors of [112] claimed that the processor thermal effect is one of the most contributing factors to the energy variation, and the CPU temperature and the energy consumption variation are tightly coupled (up to 16% increase in the variation when the temperature changed from 37.7°C to 74.5°C).

This exposes the processor temperature as a delicate factor to consider while comparing energy consumption variations across a set of homogeneous processors.

The ambient temperature was also discussed in many papers as a candidate factor for the energy variation of a processor. In [108], the authors claimed that energy consumption may vary due to fluctuations caused by the external environment. These fluctuations may alter the processor's temperature and energy consumption. However, the temperature inside a data center does not show major variations from one node to another. In [34], El Mehdi Dirouri *et al.* showed that switching the spot of two servers does not affect their energy consumption. Moreover, changing hardware components, such as the hard drive, the memory or even the power supply, does not affect the energy variation of a node, making it mainly related to the processor. This result was recently assessed by [112], where the rack placement and power supply introduced a maximum of 2.8 % variation in the observed energy consumption.

Beyond hardware components, the accuracy of power meters has also been questioned. Inadomi et al. [52] used three different power measurement tools: RAPL, Power Insight[11] and BGQ EMON. All of the three tools recorded the same 10 % of energy variation, that was supposedly related to the manufacturing process.

2.4.2 Mitigating Energy Variations

Acknowledging the energy variation problem on processors, some papers proposed contributions to reduce and mitigate this variation. In [52], the authors introduced a variation-aware algorithm that improves application performance under a power constraint by determining module-level (individual processor and associated DRAM) power allocation, with up to 5.4× speedup. The authors of [46] proposed parallel algorithms that tolerate the variability and the non-uniformity by decoupling per process communication over the available CPU. Acun *et al.* [4] found a way to reduce the energy variation on Ivy Bridge and Sandy Bridge processors, by disabling the Turbo Boost feature to stabilize the execution time over a set of processors. They also proposed some guidelines to reduce this variation by replacing the old—slower—chips, by load balancing the workload on the CPU cores and leaving one core idle. They claimed that the variation between the processor cores is insignificant. In [23], the researchers showed how a parallel system can be used to deal with the energy variation by compensating for the uneven effects of power capping.

In [75], the authors highlight the increase of energy variation across the latest Intel micro-architectures by a factor of 4 from Sandy Bridge to Broadwell, a 15 % of run-to-run variation within the same processor and the increase of the inter-cores variation from 2.5 % to 5 % due to hardware-enforced constraints, concluding with some recommendations for Broadwell usage, such as running one hyper-thread per core.

[11] https://www.itssolution.com/products/trellis-power-insight-application

2.5 Energy Optimizations

,

We will now turn our focus to the energy optimization challenge after considering the various ways of measuring energy consumption in computers and understanding the energy variation problem. Over the previous decade, there has been a considerable increase in interest in this field, with many papers proposing different approaches to reduce the energy consumption of software applications. This section will pass through the main contributions in this field, and will focus on the two following parts.

2.5.1 Energy Optimization in the Design Phase

The first part of this section will focus on energy optimizations in the conception phase, where the goal is to make the final product use less energy by choosing the best set of programming languages, tools, libraries, etc. It also includes all the work and optimizations that developers can do to the source code to make the software use less energy when it is running.

We start with the work of Pereira et al. where they did an energy consumption comparison analysis of the most popular programming languages. Using the Pareto optimum, the paper provides recommendations on how to combine some of these languages to enhance code quality while taking into account execution time, memory utilization, and energy consumption using the Pareto optimum. Some of the study's findings demonstrate that interpreted languages, such as Python, have lower energy efficiency than compiled languages, such as C or Rust. The research also offers language combinations that developers might use together to improve energy economy, execution speed, and memory utilization. This paper's findings were based on the game benchmark, which is the most famous set of benchmarks that compares several programming languages.

In this paper, we define a ranking of energy efficiency in programming languages. We considered a set of computing problems implemented in ten well-known programming languages and monitored the energy consumed when executing each language. Our preliminary results show that although the fastest languages tend to be the lowest energy-consuming ones, there are other interesting cases where slower languages are more energy-efficient than faster ones.

Couto et al. investigated the influence of programming language choice on the energy consumption of software during execution. In their paper [29], they examined a set of computing problems written in ten well-known programming languages while observing the energy required when running each language. They also found that there are interesting situations in which slower languages use less energy than faster ones, even though fast languages

are usually the ones that use the least energy. Finally, they produced an energy efficiency rating of programming languages. The paper [18] compared the energy, performance and database response time of web applications written in Java versus those written in Kotlin. They discovered that there is no statistically significant difference in CPU load between individual measurements(less than 2%) 2. However, Kotlin implementation has never earned the best results in any collection of measurements.

Other works that have studied the impact of website technology on energy consumption, [93, 74]. In their work [93],Philippot et al. measured the computer resources used during the loading of a website in a browser, such as memory utilization and energy consumption, for over 500 websites and proposed some best practices for developers.

Other works examined software energy consumption efficiency through source code changes and optimizations. For example, some papers [42, 122] studied the effect of Java collections on energy consumption, depending on the collection size and/or the most executed tasks on the collection (insertion, removal, search). They provided some insights into the energy efficiency of some collections for multiple scenarios. For example, Hasan et al. [52] compared the energy consumption of several Java data structures, analyzing the bytecode using the Wala framework8 and assessing the evolution of the energy consumption in different scenarios (insertion at the beginning, iteration, etc.). They also used some automated replacements of LinkedList and ArrayList to simulate best-and worst-case energy consumption scenarios in real production applications. Their study showed that using inappropriate collection can cause up to 300% of energy consumption inefficiency.

As for the impact of the source code on the energy consumption, we can cite [95, 36] where the authors investigated the impact of Java collections on energy usage based on collection size and its usage (insertion, removal, search) and They provide some insights into the energy efficiency of specific collections under various scenarios.

Hasan et al. examined the energy consumption of multiple Java data structures, analyzing the bytecode and evaluating the change in energy consumption in various circumstances(research, insertion, deletion, etc.). They also simulated best- and worst-case energy usage scenarios in real-world production systems by replacing the LinkedList and ArrayList, and discovered that incorrect collection can result in a 300% increase in energy consumption [48].

Other studies [72, 63] looked into the energy use of Java primitive types, string operations, and the use of exceptions, loops, and arrays. Kumar et al., for example, examined the energy usage of code snippets and micro-benchmarks and provided several conclusions, such as string concatenation would use less energy than StringBuilder and StringBuffer and static variables tends to consume 60% more energy than instance variables.

Pereira et al. [91] presented SPELL, a tool that helps developers spot energy leaks in their source code. Using a statistical spectrum-based analysis and JRAPL [42, 71], the tool locates energy-inefficient code fragments. According to the authors, it is language and context-independent.

2.5.2 Energy Optimizations in the Execution Phase

The second part of this section will focus on energy optimization in the execution phase, where the goal is to optimize this energy consumption for already developed software without making changes to the source code. The goal is to set up and create an environment in which software can run with the least amount of energy. This could involve process scheduling, system tunning and so on.

We begin with Aequitas [99], a system that allows parallel applications to live on power domains that are co-managed (sharing the same CPU). The technology is founded on the premise that coexisting programs can regard power-management hardware as a shared resource and collaborate on power management decisions. As a result, it accomplishes its purpose by scheduling these applications with a time slicing technique (as an example round robin). The authors claim that their strategy achieves a 12.9% improvement while incurring only a 2.5% performance cost.

As for virtual machines, Kurpicz et al.. analyzed the total energy consumption of a VM in a data center while emphasizing the static cost versus the dynamic one. In this paper [64], the authors introduced the transparent, reproducible, and predictive cost calculator model EPAVE for VM-based environments. The purpose of EPAVE is to provide the static cost of each VM on the server, which includes the air conditioning, power distribution, and the dynamic cost that is related to the VM activities.

The energy usage of virtual machine allocation and task placement has also been investigated [78].In this article, The authors propose a method for mapping workloads to virtual machines and virtual machines to the physical ones (PMs) in an energy-efficient manner. To solve the problem of high heterogeneity of activities and resources, the jobs are categorized based on their resource requirements, and then the relevant VM is found, followed by the appropriate PM where the selected VM can be deployed Using a cloud simulator, the authors claimed The suggested technique saves energy by reducing the number of active PMs while also minimizing the makespan and task rejection rate.

Besides the impact of the orchestration of virtual machine, some works have been targeting the runtime of specific programming languages. Using the TPC-DS benchmark,

Chiba et al. investigated the influence of HOTSPOT [12] and J9[13] on the performance of SQL-on-Hadoop systems (SPARK and TEZ) to reveal a three-fold disadvantage that one JVM can have over the other [24]. Oi also compared the performance of HOTSPOT and J9. They demonstrated in their research [84] that the relative performance of a JVM is affected by the workload. They discovered that HOTSPOT's performance ranged from 44% to 289% of J9, while its dynamic power consumption ranged from 2.7W to 7.2W, using the SPECjvm2008 [101] benchmarks.

As for python, Redondo and Ortin compared the performance and memory usage of various Python implementations (CPython, Jython, IronPython, PyPy, and so on) using a 215 set of benchmarks to discover that Python2 performed better with short applications, while Python3 versions covered more tests due to compatibility and the fact that Python2 became obsolete.[98].

2.6 Conclusion

As we have seen in the state of the art, many methods have been proposed to reduce the energy footprint of the the ICT, which they can be applied in different parts of the lifecycle of the program, from consumption to the execution. Furthermore, the execution phase took attention of many researchers because it is the part where the most energy is consumed. This thesis will focus on that aspect as well. However, unlike the most work that have been done on the hardware aspect, we will target the software impact on this energy, starting from the choice of the programming language up to how to tune some features of a frameworks in order to make the software consumes less energy. To do so we use the empirical approach due to its consistency for the moment. Unlike the performance, which is essentially related to the complexity of the algorithm, the energy consumption is more impacted by the hardware. Therefore, to optimize the energy consumption we choose a spiral method based on 3 phases:

1. execute the code,
2. measure the program,
3. infer the guidelines.

The aim of this work is to present a set of guidelines to create a benchmarking system to to measure the energy consumption of different programs. After that, we will use this system to compare the energy consumption of different programming languages. We will

[12]https://openjdk.org/groups/hotspot/
[13]https://www.eclipse.org/openj9/

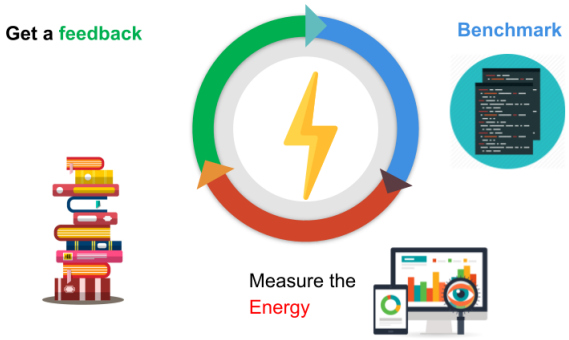


Figure 2.5: the spiral method of energy optimization

extend the work of Pereira et al. to a closer distance to production environment by comparing a set of use cases. Starting by GRPC framework, and a set of Web Frameworks. Finally, we will discuss the impact of the execution environment on the energy consumption of two of the most famous programming languages Java and Python and present how can tunning the Virtual Machine can reduce the energy consumption.

Part I

Benchmarking Programming Technologies

Chapter 3

A Benchmarking Protocol to Measure Software Energy Consumption

3.1 Introduction *TODO : missing*

This chapter covers ways to overcome empirical analysis challenges in energy consumption studies. First, we go through the three components of a successful benchmark within the energy consumption experiment. Section 3.2 concentrates on the "reproducibility" issue and how to deliver reproducible experiments without interfering with the energy measurement,while Section 3.3 discusses accuracy in software energy consumption.

3.2 Reproducibility within the context of energy

3.2.1 Introduction

Empirical measurements are critical to capture the effect of developers' choices on software energy consumption. To accomplish this, one should not overlook the benchmarking pitfalls highlighted in the state of the art [105].

Second, one should not ignore tremendous progress in computer science, which have led to a rise in the number of obsolete results. Furthermore, when it comes to comparative research, the execution environment may have an impact on the study itself. Finally, between the exploratory experiment and the publication of the results, new candidates may have emerged and others may have changed.

As a result, it is critical not only to ensure that the results can be "reproduced", so one can test their hypothesis in different environments, but also to provide room for new candidates. In this section, we will address these issues and investigate several ways of encapsulating the

systems-under-test to ensure experiment reproducibility in the context of energy consumption tests while providing the benefits and drawbacks of each strategy. Later in this section, we will also propose a protocol to ensure that the results are not only reproducible but extensible.

3.2.2 Virtual Machines

The very first choice should be to use *Virtual Machines* (VM), which allow researchers to select the most appropriate tools, software, and operating system that they are most comfortable with without incurring the cost of changing the actual working environment, giving them more control over dependencies and the execution environment. Furthermore, adopting a VM addresses the *replication crisis* since virtual images allow even the most sophisticated architecture to be simply replicated by instantiating a copy of the image.

This option, however, comes at a cost. Because of the hypervisor, software will be built on two kernels: one for the virtual machine (guest) and one for the host machine, resulting in a visible overhead and a negative influence on the performance of the system-under-test. As a result, we cannot use VM for performance-related tests. Isolation is another drawback of VM: while this feature protects the experimental setting from unwanted interference from the outside world, it is possible that this interaction is required—-especially if the experiment is dependent on an external source such as energy monitors.

3.2.3 Containers

Another option would be to use something that allows us to benefit from the host OS's isolation while simultaneously simplifying replication as proposed by VM and the direct interface with the hardware provided by the traditional techniques.

Containers provide such an advantage by ensuring application separation and ease of replication. Figure 3.1 depicts the architectural differences between virtualization and container technologies. There are three main types of virtualization.

- Type 1: runs on the hardware directly. It is primarily utilized by cloud providers where there is no host OS and only VMs that run on the open-source Xen or VMware ESX hypervisors.

- Type 2: runs on top of the host operating system and is most commonly found on personal computers. VMware server and virtualBox are notable examples of this category, and most researchers' experiments use them. However, because of the two operating systems, the applications are typically slower.

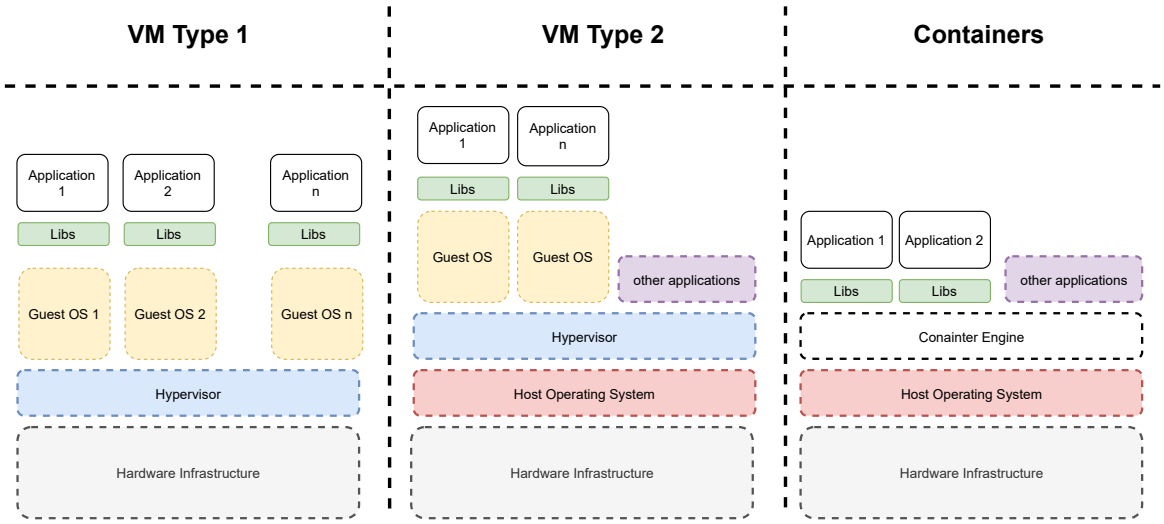


Figure 3.1: Different Methods of Virtualization

- Containers: run their operating systems on the host kernel rather than their own , which makes them smaller, faster, and more efficient in terms of hardware utilization. One can cite *Docker*, *Linux LXC*, or *LXD* [3].

3.2.4 Docker Vs. Virtual Machine

Despite the fact that Type 1 is more performant than Type 2, the latter is the most used in research, as most researchers tend to conduct their experiments on their own machines. Docker, on the other hand, is the most well-known container technology. In our case, we are more likely to promote Docker for two reasons:

1. As previously stated in literature [107, 79], we require a lightweight orchestrator to limit the overhead on energy usage of our experiments

2. We need to communicate with the host OS because we are using hardware sensors to measure the energy consumption.

3.2.5 Docker & Energy

Now that we have decided to use container technology to enclose our tests, what effect will this have on the amount of energy consumed by our tests ?

Using research from [32] who examined how adding the Docker layer affected energy consumption, Eddie Antonio Santos et al. conducted their experiment by running numerous benchmarks both with and without Docker. They contrasted the energy usage and execution

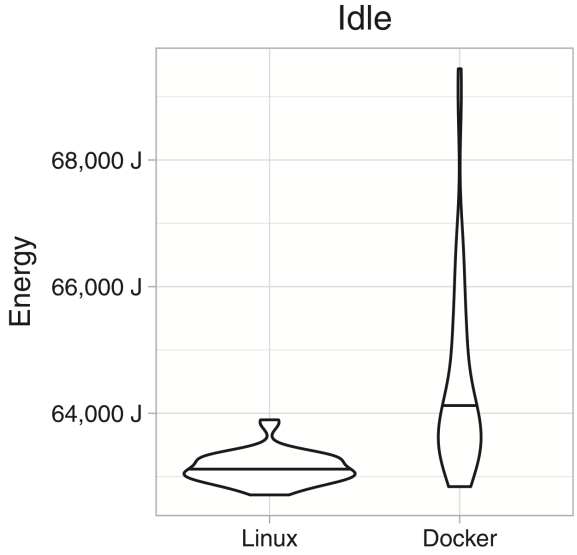


Figure 3.2: energy consumption of Idle system with and without docker [32]

time that resulted. The first step was to observe the effects of the orchestrator and the Docker deamon while there was no work to be done. Then, they use three benchmarks in their experiments: Wordpress, Reddis and PostgreSQL. The values below show the system under test's energy consumption while it is idle. Docker has an overhead of roughly $1,000$ joules, as seen in Figure 3.2 .

Yet, as seen in Figure 3.3, Docker increased the execution duration of the benchmark by 50 seconds, which led to a significant rise in the energy usage. According to the authors, this overhead is primarily caused by the Docker daemon and not by the fact that the application is in a container.

Furthermore, they calculated the cost of this extra energy, which was less than 0.15$ in the worst-case scenario, which is insignificant in comparison to the benefits of Docker for isolation and reproducibility.

To summarize, Docker-based software tends to consume more energy since it takes longer to execute. The execution of the Docker daemon causes an increase in average power consumption of only **2 Watts**. This overhead can reach up to 5% in IO-intensive applications, while it is barely visible in CPU- or DRAM-intensive workloads.

As can be seen, the introduction of the supervisor has increased the impact of Docker on energy that will be applied equitably to all experiments. Therefore, when it comes to comparison analysis, it will mitigate its impact automatically. Furthermore, because we have access to the host hardware, we do not need to worry about capturing the SUT's energy use. As a result, we will use Docker to keep all of our tests separate, as it will make sure that they can be repeated in a clear and simple way.

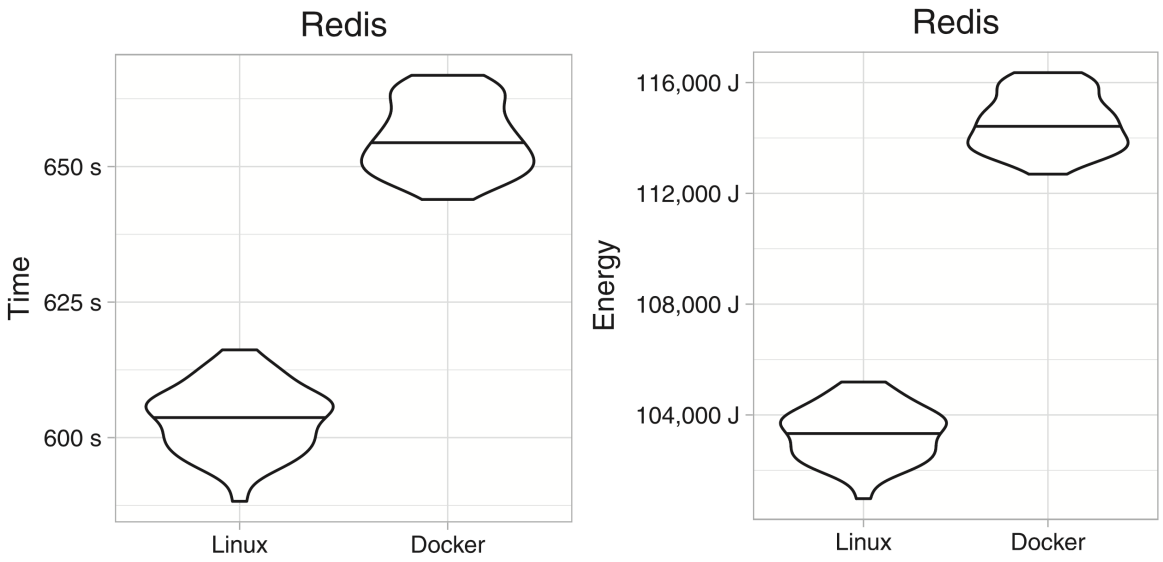


Figure 3.3: Execution time & energy consumption of Redis with and without Docker [32]

3.2.6 Extension

Definition of Extention

With the rapid evolution of the software industry, Even ensuring the reproducibility of the same research won't be enough. Each day, new versions of the software are released, and new features are added. Even more, the goal of the research might evolve. As a result, nowadays, especially in comparative studies, it is essential to leave room for expansion.

One can extand their experiment through three axes :

- **proposed solutions**: Where one will expand their research by including additional solutions and comparing them to earlier ones
- **evaluation criteria**: This axe's objective is to broaden the evaluation criteria to incorporate additional measures like as performance, cost, and security, among others.
- **benchmarks**: The purpose of this axe is to enlarge the benchmarks to include other ones in order to broaden the scope of the research.

The architecture of the extension

To be able to extend the empirical experiments through these axes, We propose to enhance the benchmarking framework suggested by the *Collaboratory on Experimental Evaluation of Software and Systems in Computer Science*[1]. Instead of only presenting the four primary

[1]http://evaluate.inf.usi.ch/

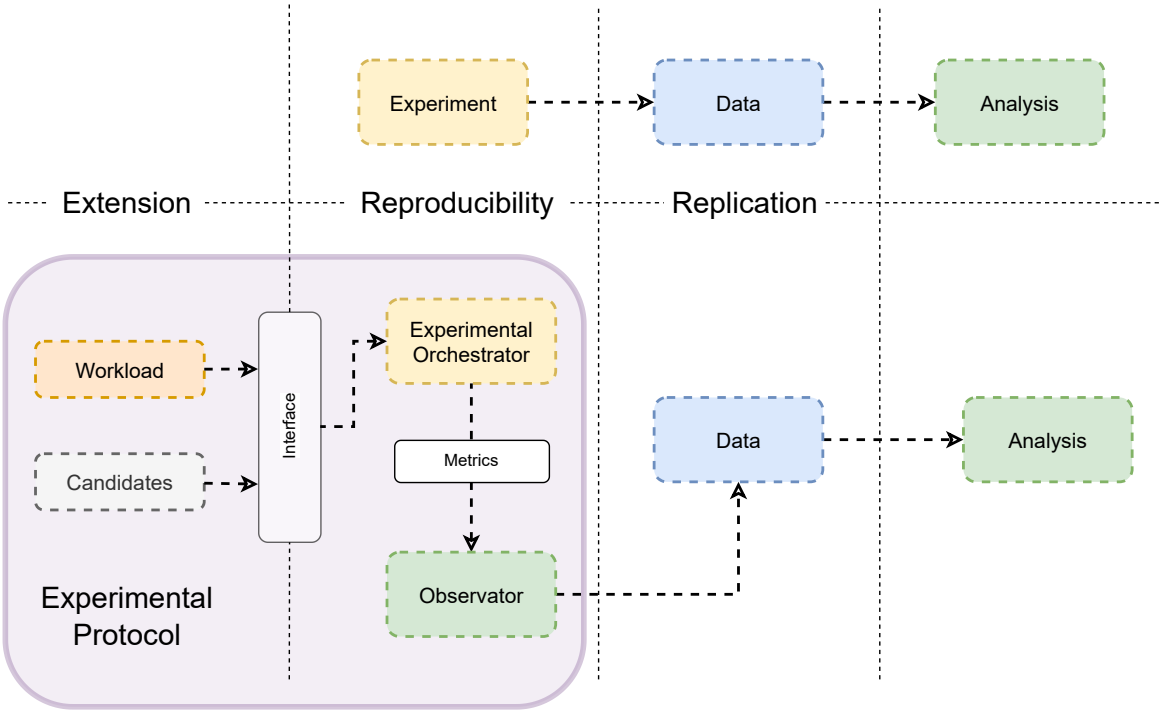


Figure 3.4: Benchmarking protocol

aspects of their guidelines that were mentioned previously in Section 3.2, we suggest an abstract model to describe an empirical experiment. Figure 3.4 shows the proposed model while providing a comparaison between the existing solution and the proposed one.

The model is composed of different components that are described below,

Context The hardware and the software configuration for the actual experiment, the purpose of this part is to provide extra information to help readers have a better judgement on the results and to be able reproduce the experiment while diminishing the impact of external factors .

orchestrator The core design of the experiment which is responsible for running the experiment regrardless of the context. In the experiment,it is the only component that is allowed to interact with the SUT. The orchestrator provides three interfaces:

- *Workload interface* : provides a set of functions that should be implemented by the workload and called by each candidate in the experiment.This interface is responsible for extending the experiment with new benchmarks.

- *The Observer interface*: provides a set of metrics that are collected by the orchestrator during the experiment. Implementing the observer interface allows the user to extend the experiment with new metrics.
- *The Candidate interface* : It provides a set of functions that should be implemented by the candidate and called by the orchestrator. This interface is responsible for extending the experiment with new solutions.

Data The row data collected collected by the experiment. In ourcase it will be provided by the observator, one ore more observators can be used to collect different metrics. The purpose of this data is to ensure the *replication* of the experiment which will allow the reader to perform extra analysis without rexucuting the experiment.

Analysis The final part, that should provide the set of methodes and functions used to do the empirical analysis in order to answer the research questions.

3.2.7 Conclusion

In this section, we addressed the first challenge of empirical research, which is the **reproducibility** of the experiments. We started by listing the different options for encapsulating the system-under-test. Then, we have shown that the use of VMs is not suitable for performance-related tests nor energy-related ones, while containers are a good alternative. Laterm, we discussed the pros and cons of using Docker for experiments related to energy consumption and have shown that it has a constant overhead which is self-mitigated when it comes to comparison analysis. Lastly, we have proposed an addition to the benchmarking framework that would allow the experiment to be extended along the three axes already mentioned.

3.3 Improving Accuracy by Taming Energy Variations

3.3.1 introduction

While the previous section aimed to ensure the reproducibility of our software energy consumption experiments, this section provides a collection of tips and tools to help increase the accuracy of these experiments. We are well aware of the effect that hardware has on energy fluctuations. However, we feel that there is still an opportunity for practitioners to minimize this energy variance by employing solely tuneable factors. To that end, we conducted a series of empirical studies utilizing state-of-the-art recommendations to discover which controllable elements can limit the variations of benchmark energy usage.

3.3.2 Research questions

This study will focus on the following research questions:

RQ 1: Does the benchmarking protocol affect the energy variation?

RQ 2: How important is the impact of the processor features on the energy variation?

RQ 3: What effect does the operating system have on energy variation?

RQ 4: Does the choice of processor make a difference in reducing the energy variation?

3.3.3 Experimental Setup

This section describes our detailed experimental environment, covering the cluster configuration and the benchmarks we used to justify our experimental methodology.

Measurement Context

There are three main contexts.

- different machines with different settings;
- different machines with the same settings ;
- the same machine.

We used the platform Grid5000 (G5K) [8, 76], which is a large, flexible testbed for experiment-driven research that is spread out across all of France, to meet these needs. Grid5000 has several clusters that are made up of 4 to 124 identical machines with different

configurations for each cluster. For our experiment, we looked at four groups. Our main criterion was the CPU configuration. Table 3.1 below, presents a description of the four clusters that have been chosen for our experiments.

Table 3.1: Description of clusters included in the study

Cluster	Processor	Nodes	RAM
Dahu	2× Intel Xeon Gold 6130	32	192 GiB
Chetemi	2× Intel Xeon E5-2630v4	15	768 GiB
Ecotype	2× Intel Xeon E5-2630Lv4	48	128 GiB
Paranoia	2× Intel Xeon E5-2660v2	8	128 GiB

As most of the nodes are equipped with two sockets (physical processors), we use the acronym CPU or socket to designate one of the two sockets and PU for the *processing unit*. For our study we consider hyper-threads as distinct **PUs**.

Figure 3.5 shows the detailed topology of a node in the cluster Dahu as an example.

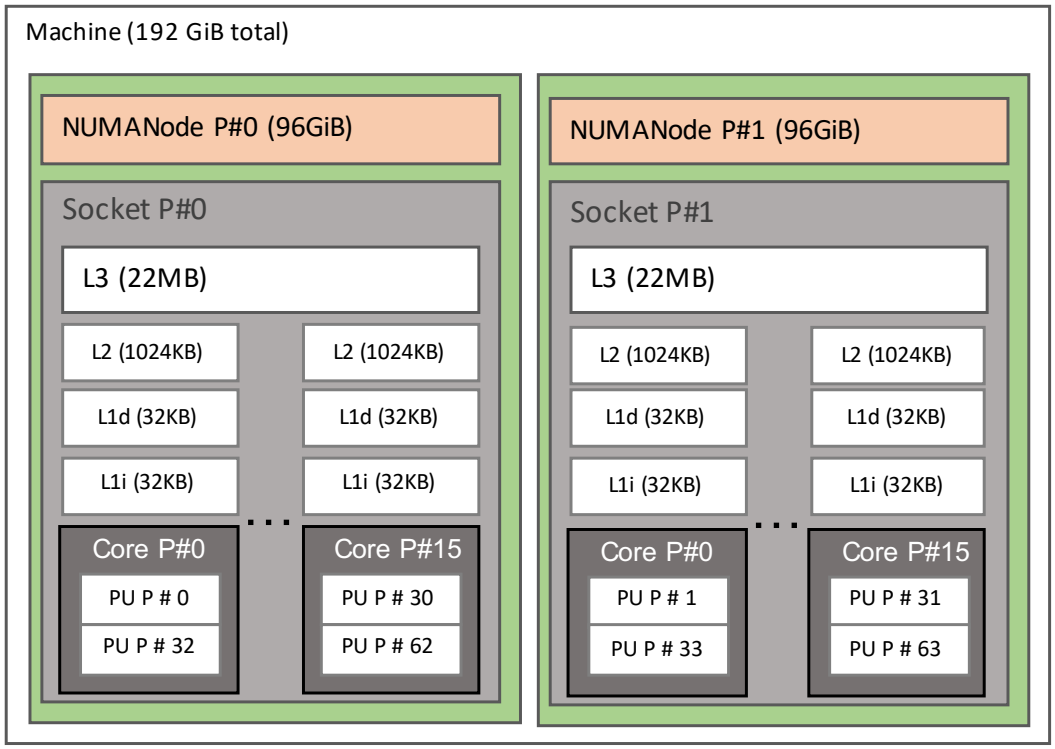


Figure 3.5: Topology of the nodes of the cluster Dahu

Workload

We picked the benchmarks based on two criteria.

First, **scalability**: We wanted to learn as much as possible about the experiment in the time we had, so we needed some benchmarks that could grow or shrink with the number of PUs and work in different situations. The second criterion is whether or not the workload is **representative**. As stated in the challenges, a workload must represent the production environment or the experiment would be inconsistent [11].

To meet these criteria, we looked at the "state of the art" and found the most common benchmarks used to test the performance of hardware. We then chose the ones that could be scaled up. Our candidate is `NAS Parallel Benchmark` (NPB v3.3.1) [7]: one of the most used benchmarks for *HPC*. We used the applications (LU), the *Conjugate Gradient* (CG) and *Embarrassingly Parallel* (EP) computation-intensive benchmarks in our experiments, with the C data class. Furthermore we have used other applications to validate our results using more applications such as `Stress-ng v0.10.0`,[2] `pbzip2 v1.1.9`,[3] `linpack`[4] and `sha256 v8.26`.[5]

Metrics & Measurement Tools

Our metric for the accuracy of the test is the Standard deviation aka **STD** of the energy consumption. Therefore whether the tests consume more or less energy is out of our scope. To study this variation we need first a tool to measure the energy consumption. For this we used POWERAPI [28], which is a power monitoring toolkit that is based on *Intel Running Average Power Limit* (RAPL) [59]. The advantage of PowerAPI is that it reports the Energy consumption of CPU and DRAM at a socket level.

Our testbeds are run with a minimal version of Debian 9 (4.9.0 kernel version)[6] where we install Docker (version 18.09.5), which will be used to run the RAPL sensor and the benchmark itself. The energy sensor collects RAPL reports and stores them in a remote MONGODB instance, allowing us to perform *post-mortem* analysis in a dedicated environment. Using Docker makes the deployment process easier on the one hand, and provides us with a built-in control group encapsulation of the conducted tests on the other hand. This allows POWERAPI to measure all the running containers, even the RAPL sensor consumption, as it is isolated in a container.

Every experiment is conducted on 100 iterations, on multiple nodes and using the 3 NPB benchmarks we mentioned, with a warmup phase of 10 iterations for each experiment. In most cases, we were seeking to evaluate the *STandard Deviation* (STD), which is the most

[2]https://kernel.ubuntu.com/~cking/stress-ng
[3]https://launchpad.net/pbzip2/
[4]http://www.netlib.org/linpack
[5]https://linux.die.net/man/1/sha256sum
[6]https://github.com/grid5000/environments-recipes/blob/master/debian9-x64-min.yaml

representative factor of the energy variation. We tried to be very careful, while running our experiments, not to fall in the most common benchmarking "crimes" [106]. As we study the STD difference of measurements we observed from empirical experiments, we use the bootstrap method [33] to randomly build multiple subsets of data from the original dataset, and we draw the STD density of those sets, as illustrated in Figure 3.6. Given the space constraints, we report on aggregated results for nodes, benchmarks and workloads. However, the raw data we collected is available through the public repository we published.[7] We believe this can help to achieve better and more reliable comparisons. We mainly consider 3 different workloads in our experiments: single process, 50 %, and 100 %, to cover the low, medium and high CPU usage when analyzing the studied parameters effect, respectively. These workloads reflect the ratio of used PU count to the total available PU.

3.3.4 Analysis

In this part, we aim to establish experimental guidelines to reduce the CPU energy variation. We therefore explore many potential factors and parameters that could have a considerable effect on the energy variation.

3.3.5 Docker & Accuracy

As the state of the art assesses the impact of Docker on the energy consumption, one can also consider its impact on accuracy. In other words:
RQ: does Docker affect the energy variation of the experiments?

To answer this question we conducted a preliminary experiment by running the same benchmarks LU, CG and EP in a Docker container and a flat binary format on 3 nodes of the cluster Dahu to assess if Docker induces an additional variation. Figure 3.6 reports that this is not the case, as the energy consumption variation does not get noticeably affected by Docker while running a same compiled version of the benchmarks at 5 %, 50 % and 100 % workloads. In fact, while Docker increases the energy consumption due to the extra layer it implements [32], it does not noticeably affect the energy variation. The *standard deviation* (STD) is even slightly smaller ($STD_{Docker} = 192mJ$,$STD_{Binary} = 207mJ$), taking into account the measurements errors and the OS activity.

[7] https://github.com/anonymous-data/Energy-Variation

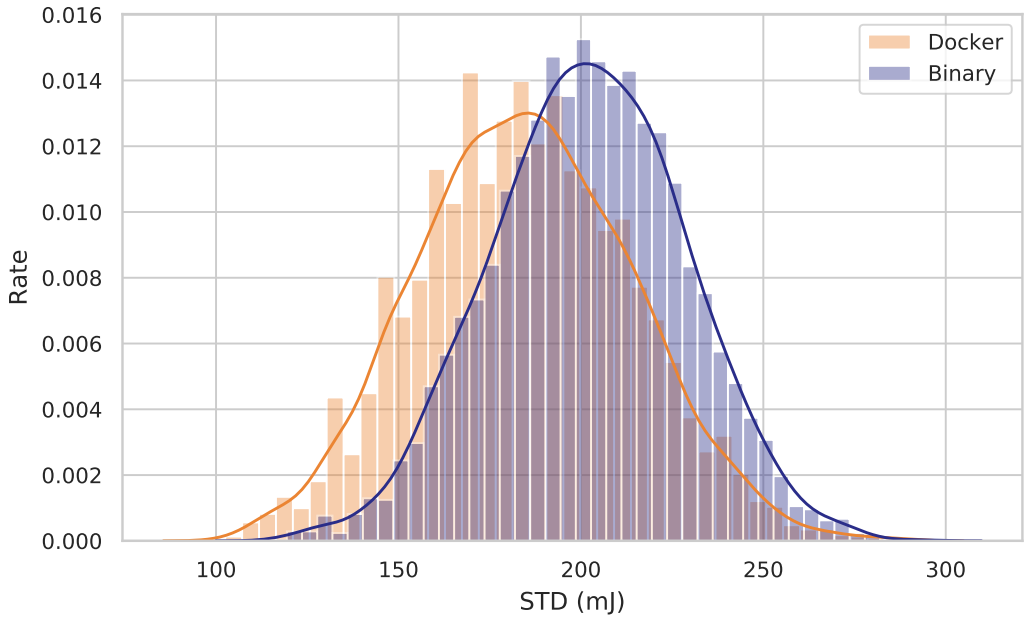


Figure 3.6: Comparing the variation of binary and Docker versions of aggregated LU, CG and EP benchmarks

RQ 1: Benchmarking Protocol

To achieve a robust and reproducible experiment, practitioners often tend to repeat their tests multiple times, in order to analyze the related performance indicators, such as execution time, memory consumption or energy consumption. We therefore aim to study the benchmarking protocol to identify how to efficiently iterate the tests to capture a trustable energy consumption evaluation.

In this first experiment, we investigate if changing the testing protocol affects the energy variation. To achieve this, we considered 3 execution modes: In the "normal" mode, we iteratively run the benchmark 100 times without any extra command, while the "sleep" mode suspends the execution script for 60 seconds between iterations. Finally, the "reboot" mode automatically reboots the machine after each iteration. The difference between the normal and sleep modes intends to highlight that the CPU needs some rest before starting another iteration, especially for an intense workload. Putting the CPU into sleep for several seconds could give it some time to reach a lower frequency state or/and reduce its temperature, which could have an impact on the energy variation. The reboot mode, on the other hand, is the most straightforward way to reset the machine state after every iteration. It could also be beneficial to reset the CPU frequency and temperature, the stored data, the cache or the CPU

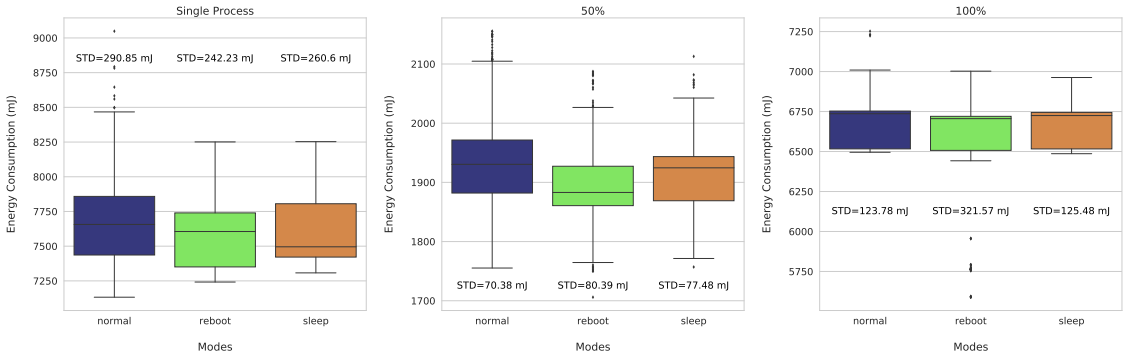


Figure 3.7: Energy variation with the normal, sleep and reboot modes

registries. However, the reboot task takes a considerable amount of time, so rebooting the node after every single operation is not the fastest nor the most eco-friendly solution, but it deserves to be checked to investigate if it effectively enhances the overall energy variation or not.

Figure 3.7 reports on 300 aggregated executions of the benchmarks LU, CG and EP, on 4 machines of the cluster Dahu (cf. Table 3.1) for different workloads. We note that the results have been executed with different datasets sizes (B, C and D for single process, 50 % and 100 % respectively) to remedy to the brief execution times at high workloads for small datasets. This justifies the scale differences of reported energy consumptions between the 3 modes in Figure 3.7. As one can observe, picking one of these strategies does not have a strong impact on the energy variation for most workloads. In fact, all the strategies seem to exhibit the same variation with all the workloads we considered—*i.e.*, the STD is tightly close between the three modes. The only exception is the reboot mode at 100 % load, where the STD is 150 % times worst, due to an important amount of outliers. This goes against our expectation, even when setting a warm-up time after reboot to stabilize the OS.

In Figure 3.8, we study the standard deviation of the three modes by constituting $5,000$ random 30-iterations sets from the previous executions set and we compute the STD in each case, considering mainly the 100 % workload as the STD was 150 % higher for the reboot mode with that load. We can observe that the considerable amount of outliers in the reboot mode is not negligible, as the STD density is clearly higher than the two other modes. This makes the reboot mode as the less appropriate for the energy variation at high workloads.

To answer RQ 1, we conclude that the benchmarking protocol **partially affects** the energy variation, as highlighted by the reboot mode results for high workloads.

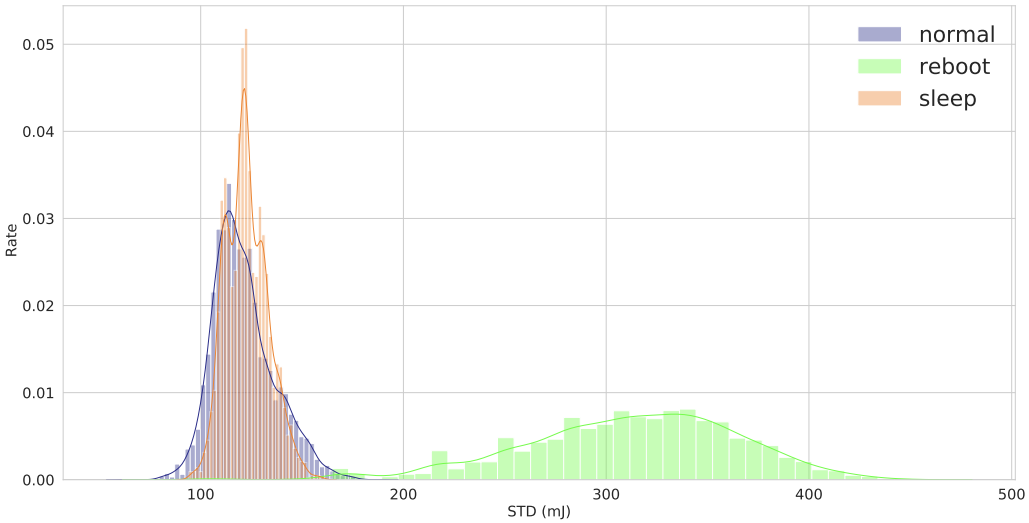


Figure 3.8: STD analysis of the normal, sleep and reboot modes

RQ 2: Processor Features

The C-states provide the ability to switch the CPU between more or less consuming states upon activities. Turning the C-states on or off have been subject of many discussions [109], because of its dynamic frequency mechanism but, to the best of our knowledge, there have been no fully conducted C-states behavior analysis on CPU energy variation.

We intend to investigate how much the energy consumption varies when disabling the C-states (thus, keeping the CPU in the C0 state) and at which workload. Figure 3.9 depicts the results of the experiments we executed on three nodes of the cluster Dahu. On each node, we ran the same set of benchmarks with two modes: C-states on, which is the default mode, and C-states off. Each iteration includes 100 executions of the same benchmark at a given workload, with three workload levels. We note that our results have been confirmed with the benchmarks LU, CG and EP.

We can clearly see the effect that has the C-states off mode when running a single-process application/benchmark. The energy consumption varies 5 times less than the default mode. In this case, only one CPU core is used among 2×16 physical cores. The other cores are switched to a low consumption state when C-states are on, the switching operation causes an important energy consumption difference between the cores, and could be affected by other activities, such as the kernel activity, causing a notable energy consumption variation. On the other hand, switching off the C-states would keep all the cores—even the unused ones—at a

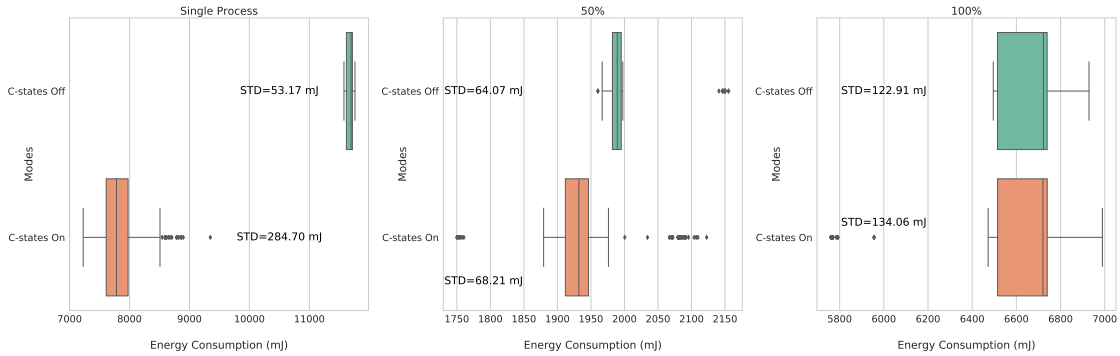


Figure 3.9: Energy variation when disabling the C-states

high frequency usage. This highly reduces the variation, but causes up to 50 % of extra energy consumption in this test ($Mean_{C-states-off} = 11,665mJ$,$Mean_{C-states-on} = 7,641mJ$).

At a 100 % workload, disabling the C-states seems to have no effect on the total energy consumption nor its variation. In fact, all the cores are used at 100 % and the C-states module would have no effect, as the cores are not idle. The same reason would apply for the 50 % load, as the hyper-threading is active on all cores, thus causing the usage of most of them. For single process workloads, disabling the C-states causes the process to consume 50 % more energy as reported in Figure 3.9, but reduces the variation by 5 times compared to the C-states on mode. This leads to mainly two questions: Can a process pinning method reduce/increase the energy variation? And, how does the energy consumption variation evolve at different PU usage level?

Cores Pinning To answer the first question, we repeated the previous test at 50 % workload. In this experiment, we considered three cores usage strategies, the first one (S1) would pin the processes on all the PU of one of the two sockets (including hyper-threads), so it will be used at 100 %, and leave the other CPU idle. The second strategy (S2) splits the workload on the two sockets so each CPU will handle 50 % of the load. In this strategy, we only use the core PU and not the hyper-threads PU, so every process would not share his core usage (all the cores are being used). The third strategy (S3) consists also on splitting the workload between the two sockets, but considering the usage of the hyper-threads on each core—*i.e.*, half of the cores are being used over the two CPU. Figure 3.10 reports on the energy consumption of the three strategies when running the benchmark CG on the cluster Dahu. We can notice the big difference between these three execution modes that we obtained only by changing the PU pinning method (that we acknowledged with more than 100 additional runs over more than 30 machines and with the benchmarks LU and EP). For example, S2 is the least power

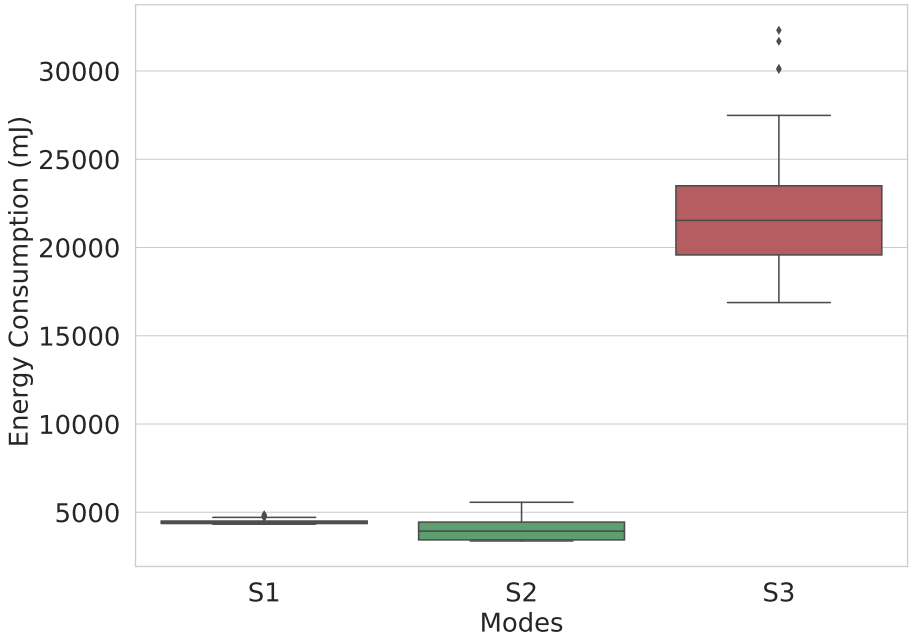


Figure 3.10: Energy variation considering the three cores pinning strategies at 50 % workload

consuming strategy. We argue that the reason is related to the isolation of every process on a single physical core, reducing the context switch operations. In the first and third strategy, 32 processes are being scheduled on 16 physical cores using the hyper-threads PU, which will introduce more context switching, and thus more energy consumption.

We note that even if the first and third strategies are very similar (both use hyper-threads, but only on one CPU for the first and on two CPU for the third), the gap between them is considerable variation-wise, as the variation is 30 times lower in the first strategy ($STD_{S1} = 116mJ$,$STD_{S3} = 3,452mJ$). This shows that the usage of the hyper-threads technology is not the main reason behind the variation, the first strategy has even less variation than the second one and still uses the hyper-threading.

The reason for the S1 low energy consumption is that one of the two sockets is idle and will likely be in a lower power P-state, even with the disabled C-states. The S2 case is also low energy consuming because by distributing the threads across all the cores, it completes the task faster than in the other cases. Hence, it consumes less energy. The S3 is a high consuming strategy because both sockets are being used, but only half the cores are active. This means that we pay the energy cost for both sockets being operational and for the experiments taking longer to run because of the recurrent context switching.

Table 3.2: STD (mJ) comparison for 3 pinning strategies

Strategy	**S1**	**S2**	**S3**
Node 1	88	270	1,654
Node 2	79	283	2,096
Node 3	58	287	1,725
Node 4	51	229	1,334

Our hypothesis regarding the worst results that we observed when using the third strategy is the recurrent context switching, added to the OS scheduling that could reschedule processes from a socket to another, which invalids the cache usage as a process can not take profit of the socket local L3 cache when it moves from a CPU to another (cf. Figure 3.5).

Moreover, the fact that the variation is 4–5 times higher when using the strategy S2 compared to S1 ($STD_{S1} = 116mJ$, $STD_{S3} = 575mJ$), gives another reason to believe that swapping a process from a CPU to another increases the variation due to CPU micro differences, cache misses and cache coherency. While the mean execution time for the strategy S3 is very high ($MeanTime_{S3} = 46s$) compared to the two other strategies ($MeanTime_{S1} = 11s$, $MeanTime_{S2} = 7s$), we see no correlation between the execution time and the energy variation, as the S1 still give less variations than S2 even if it takes 36 % more time to run.

Table 3.2 reports on additional aggregated results for the STD comparison on four other nodes of the cluster Dahu at 50 %, with the benchmarks LU, CG and EP. In fact, the CPU usage strategy S1 is by far the experimentation mode that gave the least variation. The STD is almost 5 times better than the strategy S2, but is up to 10 % more energy consuming ($Mean_{S1} = 4469mJ$, $Mean_{S2} = 4016mJ$). On the other hand, the strategy S3 is the worst, where the energy consumption can be up to 5 times higher than the strategy S2 ($Mean_{S2} = 4016mJ$, $Mean_{S3} = 21645mJ$) and the variation is much worst (30 times compared to the first strategy). These results allow us to have a better understanding of the different processes-to-PU pinning strategies, where isolating the workload on a single CPU is the best strategy. Using the hyper-threads PU on multiple sockets seems to be a bad recommendation, while keeping the hyper-threading enabled on the machine is not problematic, as long as the processes are correctly pinned on the PU. Our experiments show that running one hyper-thread per core is not always the best to do, at the opposite of the claims of [75].

Processes Threshold To answer the second question regarding the evolution of the energy variation at different levels of CPU usage, we varied the used PU's count to track the EV evolution. Figure 3.11 compares the aggregated energy variation when the C-states are on and off using 2, 4 and 8 processes for the benchmarks LU, CG and EP. This figure confirms that

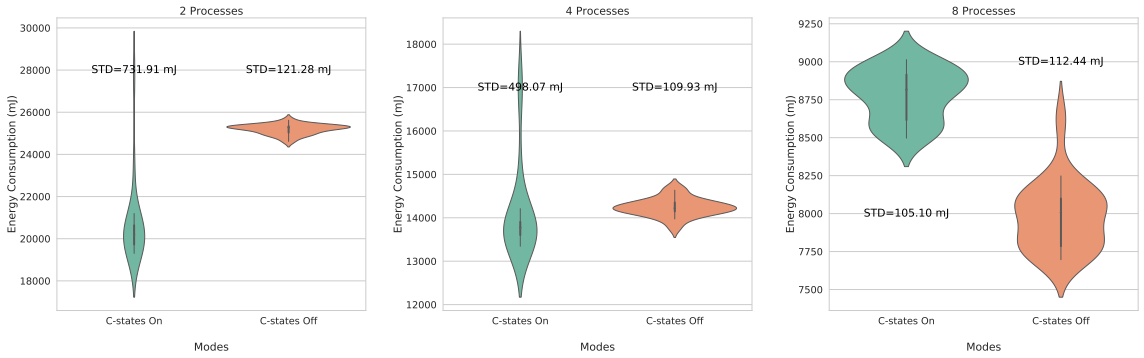


Figure 3.11: C-states effect on the energy variation, regarding the application processes count

disabling the CPU C-states does not decrease the variation for all the workloads, as we can clearly observe, the variation is increasing along with the number of processes. When running only 2 processes, turning off the C-states reduces the STD up to 6 times, but consumes 20 % more energy ($Mean_{C-states-on} = 10,334mJ$, $Mean_{C-states-off} = 12,594mJ$). This variation is 4 times lower when running 4 processes and almost equal to the C-states on mode when running 8 processes. In fact, running more processes implies to use more CPU cores, which reduces the idle cores count, so the cores will more likely stay at a higher consumption state even if the C-states mechanism is on.

In our case, using 4 PU reduces the variation by 4 times and consumes almost the same energy as keeping the C-states mechanism on ($Mean_{C-states-on} = 7,048mJ$, $Mean_{C-states-off} = 7,119mJ$). This case would be the closest to reality as we do not want to increase the energy consumption while reducing the variation, but using a lower number of PU still results in less variation, even if it increases the overall energy consumption.

We note that disabling the C-states is not recommended in production environments, as it introduces extra energy consumption for low workloads (around 50 % in our case for a single process job). However, our goal is not to optimize the energy consumption, but to minimize the energy variation. Thus, disabling the C-states is very important to stabilize the measurements in some cases when the variation matters the most. Comparing the energy consumptions of two algorithms or two versions of a software systems is an example of use case benefiting from this recommendation.

Turbo Boost The Turbo Boost—also known as *Dynamic Overclocking*—is a feature that has been incorporated in Intel CPU since the Sandy Bridge micro-architecture, and is now widely available on all of the Core i5, Core i7, Core i9 and Xeon series. It automatically raises some of the CPU cores operating frequency for short periods of time, and thus boost

Table 3.3: STD (mJ) comparison when enabling/disabling the Turbo Boost

Turbo Boost	**Enabled**	**Disabled**
EP / 5 %	310	308
CG / 25 %	95	140
LU / 25 %	204	240
EP / 50 %	84	79
EP / 100 %	125	110

performances under specific constraints. When demanding tasks are running, the operating system decides on using the highest performance state of the processor.

Disabling or enabling the Turbo Boost has a direct impact on the CPU frequency behavior, as enabling it allows the CPU to reach higher frequencies in order to execute some tasks for a short period of time. However, its usage does not have a trivial impact on the energy variation. Acun *et al.* [4] tried to track the Turbo Boost impact on the Ivy Bridge and the Sandy Bridge architectures. They concluded that it is one of the main responsible for the energy variation, as it increases the variation from 1 % to 16 %. In our study, we included a Turbo Boost experiment in our testbed, to check this property on the recent Xeon Gold processors, covering various workloads.

The experiment we conducted showed that disabling the Turbo Boost does not exhibit any considerable positive or negative effect on the energy variation. Table 3.3 compares the STD when enabling/disabling the Turbo Boost, where the columns are a combination of workload and benchmark. In fact, we only got some minor measurements differences when switching on and off the Turbo Boost, and where in favor or against the usage of the Turbo Boost while repeating tests, considering multiple nodes and benchmarks. This behavior is mainly related to the *thermal design power* (TDP), especially at high workloads executions. When a CPU is used at its maximum capacity, the cores would be heating up very fast and would hit the maximum TDP limit. In this case, the Turbo Boost cannot offer more power to the CPU because of the CPU thermal restrictions. At lower workloads, the tests we conducted proved that the Turbo Boost is not one of the main reasons of the energy variation. In fact, the variation difference is barely noticeable when disabling the Turbo Boost, which cannot be considered as a result regarding the OS activity and the measurement error margin. We cannot affirm that the Turbo Boost does not have an impact on all the CPU, as we only tested on two recent Xeon CPU (clusters Chetemi and Dahu). We confirmed our experiments on these machines 100 times at 5 %, 25 %, 50 % and 100 % workloads.

We conclude that CPU features **highly impact** the energy variation as an answer for RQ 2.

RQ 3: Operating System

The *operating system* (OS) is the layer that exploits the hardware capabilities efficiently. It has been designed to ease the execution of most tasks with multitasking and resource sharing. In some delicate tests and measurements, the OS activity and processes can cause a significant overhead and therefore a potential threat to the validity. The purpose behind this experiment is to determine if the sampled consumption can be reliably related to the tested application, especially for low-workload applications where CPU resources are not heavily used by the application.

The first way to do is to evaluate the OS idle activity consumption, and to compare it to a low workload running job. Therefore, we ran 100 iterations of a single process benchmark EP, LU and CG on multiple nodes from the cluster Dahu, and compared the energy behavior of the node with its idle state on the same duration. The aggregated results, illustrated in Figure 3.12, depict that the idle energy variation is up to 140 % worst than when running a job, even if it consumes 120 % less energy ($Mean_{Job} = 8,746mJ$, $Mean_{Idle} = 3,927mJ$). In fact, for the three nodes, randomly picked from the cluster Dahu, the idle variation is way more important than when a test was running, even if it is a single process test on a 32-cores node. This result shows that OS idle consumption varies widely, due to the lack of activity and the different CPU frequencies states, but it does not mean that this variation is the main responsible for the overall energy variation. The OS behaves differently when a job is running, even if the amount of available cores is more than enough for the OS to keep his idle behavior when running a single process.

Inspecting the OS idle energy variation is not sufficient to relate the energy variation to the active job. In fact, the OS can behave differently regarding the resource usage when running a task. To evaluate the OS and the job energy consumption separately, we used the POWERAPI toolkit. This fine-grained power meter allows the distribution of the RAPL global energy across all the Cgroups of the OS using a power model. Thus, it is possible to isolate the job energy consumption instead of the global energy consumption delivered by RAPL. To do so, we ran tests with a single process workload on the cluster Dahu, and used the POWERAPI toolkit to measure the energy consumption. Then, we compared the job energy consumption to the global RAPL data. We calculated the Pearson correlation [1] of the energy consumption and variation between global RAPL and POWERAPI, as illustrated in Figure 3.13. The job energy consumption and variation are strongly correlated with the global energy consumption and variation with the coefficients 93.6 % and 85.3 %, respectively. However, this does not completely exclude the OS activity, especially if the jobs have tight interaction with the OS through the signals and system calls. This brings a new question on whether applying extra-tuning on a minimal OS would reduce the variation? As well as what

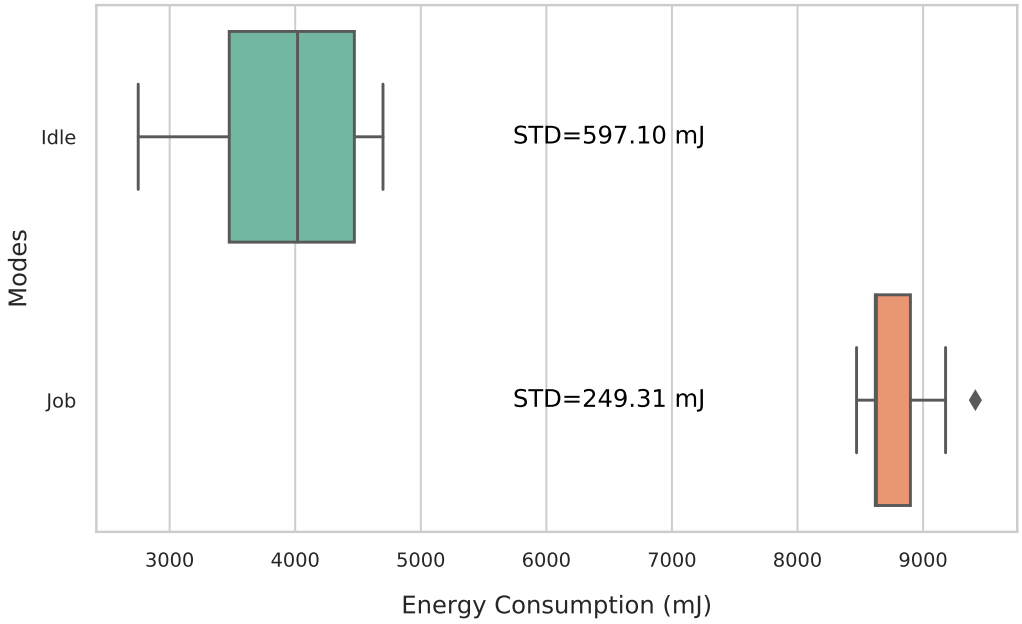


Figure 3.12: OS consumption between idle and when running a single process job

is the effect of the Meltdown security patch—that is known to be causing some performance degradation [60, 70]—on the energy variation?

OS Tuning An OS is a pack of running processes and services that might or not be required its execution. In fact, even using a minimal version of a Debian Linux, we could list many OS running services and process that could be disabled/stopped without impacting the test execution. This extra-tuning may not be the same depending on the nature of the test or the OS. Thus, we conducted a test with a deeply-tuned OS version. We disabled all the services/processes that are not essential to the OS/test running, including the OS networking interfaces and logging modules, and we only kept the strict minimum required to the experiment's execution. Table 3.4 reports on the aggregated results for running single process measurements with the benchmarks CG, LU and EP, on three servers of the cluster Dahu, before and after tuning the OS. Every cell contains the *STD* value before the tuning, plus/minus a ratio of the energy variation after the tuning. We notice that the energy variation varies less than 10 % after the extra-tuning. We argue that this variation is not substantial, as it is not stable from a node to another. Moreover, 10 % of variation is not a representative difference, due to many factors that can affect it as the CPU temperature or the measurement errors.

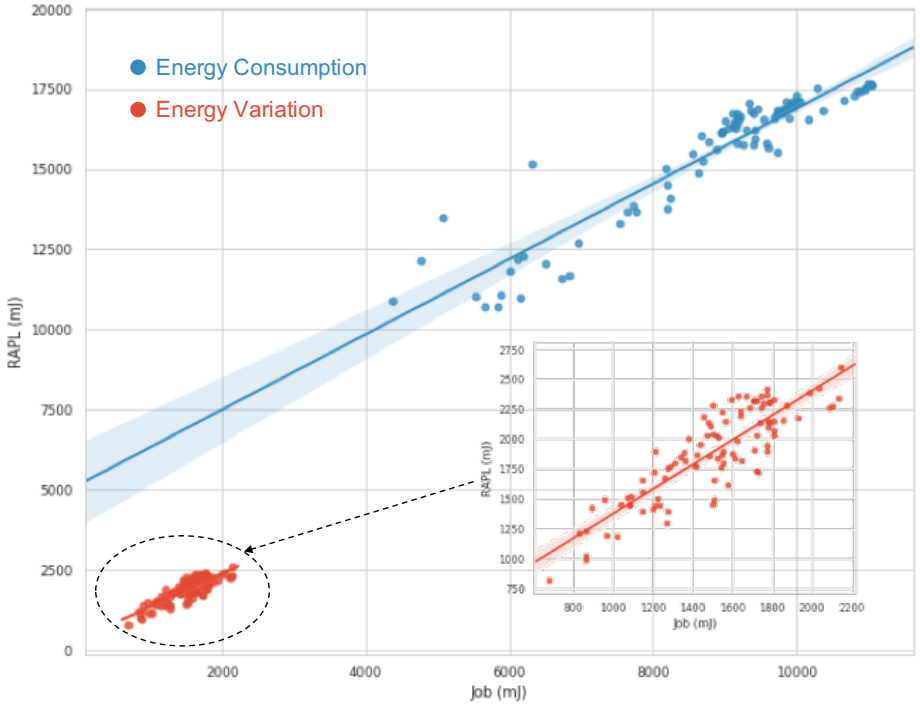


Figure 3.13: The correlation between the RAPL and the job consumption and variation

Table 3.4: STD (mJ) comparison before/after tuning the OS

Node	EP	CG	LU
N1	1370 -9 %	78 +7 %	128 +2 %
N2	1278 -7 %	64 -1 %	120 +9 %
N3	1118 +1 %	83 +2 %	93 +7 %

Speculative Executions Meltdown and Spectre are two of the most famous hardware vulnerabilities discovered in 2018, and exploiting them allows a malicious process to access others processes data that is supposed to be private [60, 70]. They both exploit the speculative execution technique where a process anticipates some upcoming tasks, which are not guaranteed to be executed, when extra resources are available, and revert those changes if not. Some OS-level patches had been applied to prevent/reduce the criticality of these vulnerabilities. On the Linux kernel, the patch has been automatically applied since the version 4.14.12. It mitigates the risk by isolating the kernel and the user space and preventing the mapping of most of the kernel memory in the user space. Nikolay *et al.* have studied in [102] the impact of patching the OS on the performance. The results showed that the overall performance decrease is around 2–3 % for most of the benchmarks and real-world applications, only some specific functions can meet a high performance decrease. In our study, we are interested in the applied patch's impact on the energy variation, as the performance decrease could mean an energy consumption increase. Thus, we ran the same benchmarks LU, CG ad EP on the cluster Dahu with different workloads, using the same OS, with and without the security patch. Table 3.5 reports on the STD values before disabling the security patch. A minus means that the energy varies less without the patch being applied, while a plus means that it varies more. These results help us to conclude that the security patch's effect on the energy variation is not substantial and can be absorbed through the error margin for the tested benchmarks. In fact, the best case to consider is the benchmark LU where the energy variation is less than 10 % when we disable the security patch, but this difference is still moderate. The little performance difference discussed in [60, 70] may only be responsible of a small variation, which will be absorbed through the measurement tools and external noise error margin in most cases.

Table 3.5: STD (mJ) comparison with/without the security patch

Node	EP	CG	LU
N1	269 +2 %	83 +1 %	108 -6 %
N2	195 +1 %	84 -5 %	121 -9 %
N3	223 +/-1 %	72 -4 %	117 +8 %
N4	276 +3 %	60 +0 %	113 -3 %

To answer RQ 3, we conclude that the OS **should not be the main focus** of the energy variation taming efforts.

Table 3.6: STD (mJ) comparison of experiments from 4 clusters

Cluster	Dahu	Chetemi	Ecotype	Paranoia
Arch	Skylake	Broadwell	Broadwell	Ivy Bridge
Freq	3.7 GHz	3.1 GHz	2.9 GHz	3.0 GHz
TDP	125 W	85 W	55 W	95 W
5%	364	210	**75**	**76**
50%	98	86	**49**	244
100%	119	116	**106**	240

RQ 4: Processor Generation

Intel microprocessors have noticeably evolved during these last 20 years. Most of the new CPU come with new enhancements to the chip density, the maximum Frequency or some optimization features like the C-states or the Turbo Boost. This active evolution caused that different generations of CPU can handle a task differently. The aim of this expriment is not to justify the evolution of the variation across CPU versions/generations, but to observe if the user can choose the best node to execute her experiments. Previous papers have discussed the evolution of the energy consumption variation across CPU generations and concluded that the variation is getting higher with the latest CPU generations [Wang et al., 75], which makes measurements stability even worse. In this experiment, we therefore compare four different generations of CPU with the aim to evaluate the energy variation for each CPU and its correlation with the generation. Table 3.6 indicates the characteristics of each of the tested CPU.

Table 3.6 also shows the aggregated energy variation of the different generations of nodes for the benchmarks LU, CG and EP. The results attest that the latest versions of CPU do not necessarily cause more variation. In the experiments we ran, the nodes from the cluster Paranoia tend to cause more variation at high workloads, even if they are from the latest generation. While the Skylake CPU of the cluster Dahu cause often more energy variation than Chetemi and the Ecotype Broadwell CPU. We argue that the hypothesis "*the energy consumption on newer CPU varies more*" could be true or not depending on the compared generations, but most importantly, the chips energy behaviors. On the other hand, our experiments showed the lowest energy variation when using the Ecotype CPU, these CPU are not the oldest nor the latest, but are tagged with "L" for their low power/TDP. This result rises another hypothesis when considering CPU choice, which implies selecting the CPU with a low TDP. This hypothesis has been confirmed on all the Ecotype cluster nodes, especially at low and medium workloads.

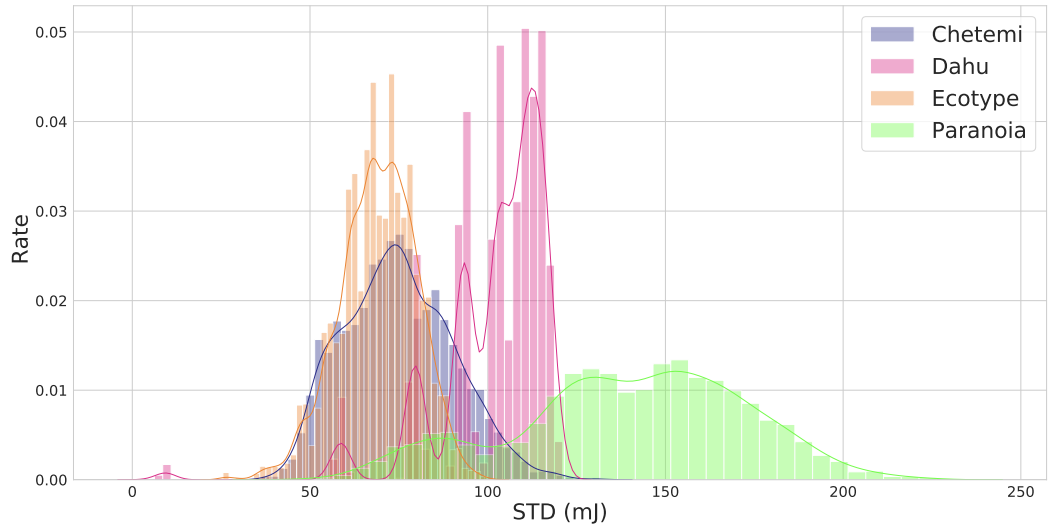


Figure 3.14: Energy consumption STD density of the 4 clusters

Figure 3.14 is an illustration of the aggregated STD density of more than 5,000-random values sets taken from all the conducted experiments. This shows that the cluster Paranoia reports the worst variation in most cases, and that Ecotype is the best cluster to consider to get the least variations, as it has a higher density for small variation values.

We conclude on **affirming RQ 4**, as selecting the right CPU can help to get less variations.

3.3.6 Experimental Guidelines

To summarize our experiments, we provide some experimental guidelines in Table 3.7, based on the multiple experiments and analysis we did. These guidelines constitute a set of minimal requirements or best practices, depending on the workload and the criticality of the energy measurement precision. It therefore intends to help practitioners in taming the energy variation on the selected CPU, and conduct the experiments with the least variations.

Table 3.7 gives a proper understanding of known factors, like the C-states and its variation reduction at low workloads. However, it also lists some new factors that we identified along the analysis we conducted in Section, such as the results related to the OS or the reboot mode. Some of the guidelines are more useful/efficient for specific workloads, as showed in our experiments. Thus, qualifying the workload before conducting the experiments can help in choosing the proper guidelines to apply. Other studied factors are not been mentioned in the

Table 3.7: Experimental Guidelines for Energy Variations

Guideline	Load	Gain
Use a low TDP CPU	Low & medium	Up to 3×
Disable the CPU C-states	Low	Up to 6×
Use the least of sockets in a case of multiple CPU	Medium	Up to 30×
Avoid the usage of hyper-threading whenever possible	Medium	Up to 5×
Avoid rebooting the machine between tests	High	Up to 1.5×
Do not relate to the machine idle variation to isolate a test EC, the CPU/OS changes its behavior when a test is running and can exhibit less variation than idle	Any	—
Rather focus the optimization efforts on the system under test than the OS	Any	—
Execute all the similar and comparable experiments on a same machine. Identical machines can exhibit many differences regarding their energy behavior	Any	Up to 1.3×

guidelines, like the Turbo Boost or the Speculative execution, due to the small effect that has been observed in our study.

In order to validate the accuracy of our guidelines among a varied set of benchmarks on one hand, and their effect on the variation between identical machines on the other hand, we ran seven experiments with benchmarks and real applications on a set of four identical nodes from the cluster Dahu, before (normal mode where everything is left to default and to the charge of the OS) and after (optimized) applying our guidelines. Half of these experiments has been performed at a 50 % workload and the other half on single process jobs. The choice of these two workloads is related to the optimization guidelines that are mainly effective at low and medium workloads. We note that we used the cluster Dahu over Ecotype to highlight the guidelines effect on the nodes where the variation is susceptible to be higher.

Figure 3.15 and 3.16 highlight the improvement brought by the adoption of our guidelines. They demonstrate the intra-node STD reduction at low and medium workloads for all the benchmarks used at different levels. Concretely, for low workloads, the energy variation is 2–6 times lower after applying the optimization guidelines for the benchmarks LU and EP, as

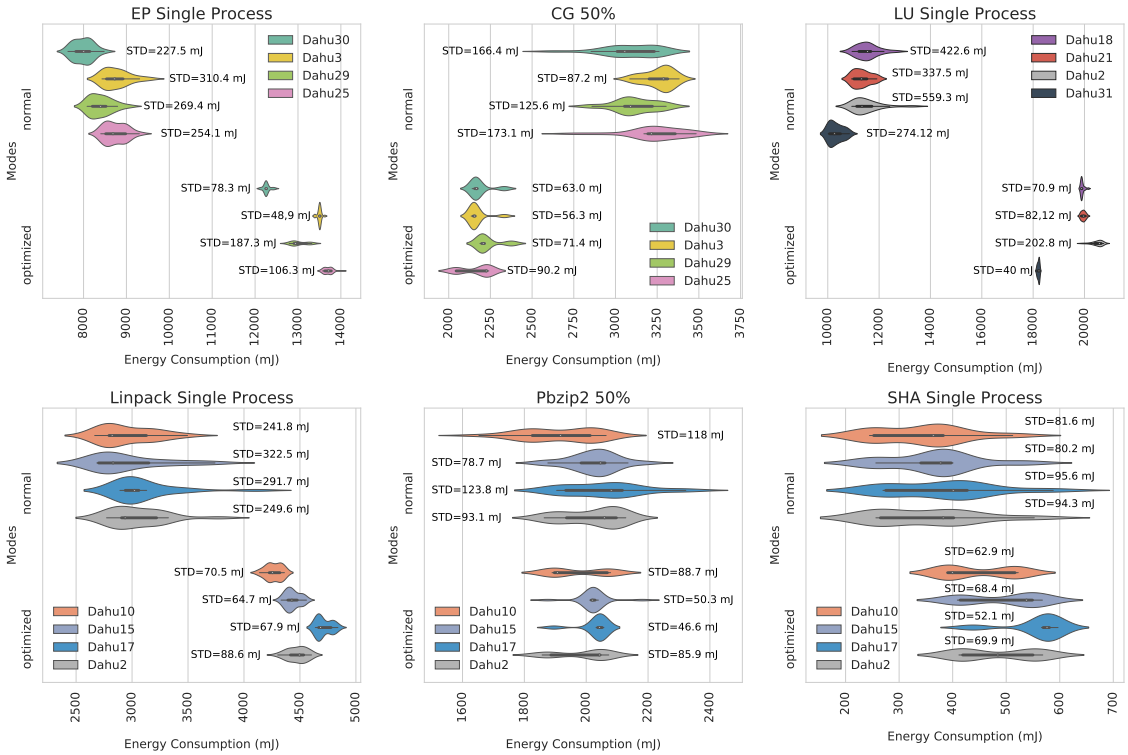


Figure 3.15: Energy variation comparison with/without applying our guidelines

well as LINPACK, while it is 1.2–1.8 times better for `Sha256`. For this workload, the overall energy consumption after optimization can be up to 80 % higher due to disabling the C-states to keep all the unused cores at a high power consumption state ($Mean_{LU-normal-Dahu2} = 11,500mJ$, $Mean_{LU-optimized-Dahu2} = 20,508mJ$). For medium workloads, the STD, and thus variation, is up to 100 % better for the benchmark CG, 20–150 % better for the `pbzip2` application and up to 100% for STRESS-NG. We note that the optimized version consumes less energy thanks to an appropriate core pinning method.

Figures 3.15 and 3.16 also highlight that applying the guidelines does not reduce the inter-nodes variation in all the cases. This variation can be up to 30 % in modern CPU [Wang et al.]. However, taming the intra-node variation is a good strategy to identify more relevant mediums and medians, and then perform accurate comparisons between the nodes variation. Even though, using the same node is always better, to avoid the extra inter-nodes variation and thus improve the stability of measurements.

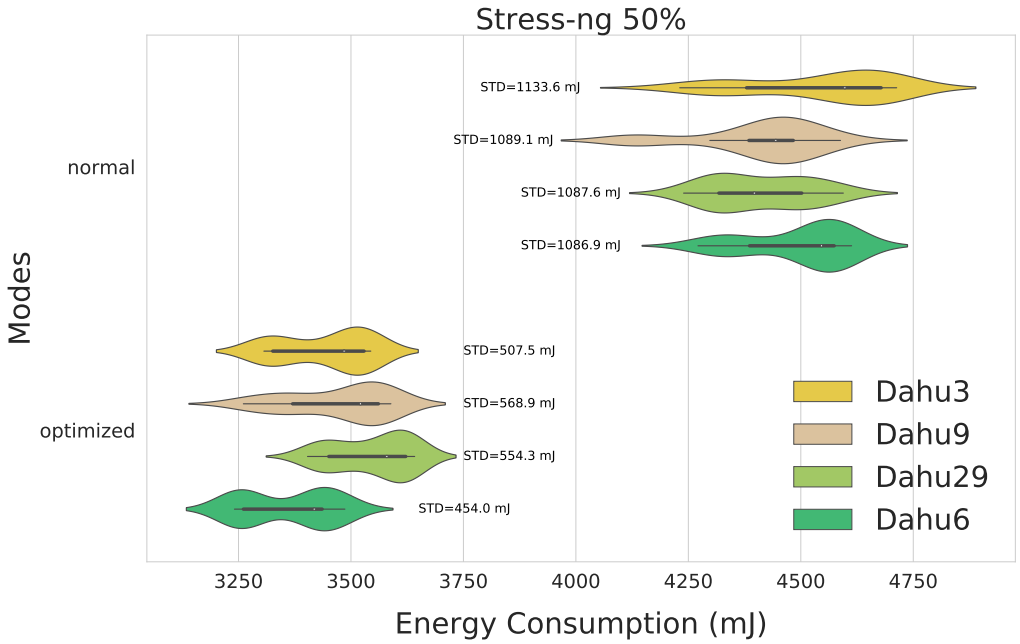


Figure 3.16: Energy variation comparison with/without applying our guidelines for STRESS-NG

3.3.7 Threats to Validity

A number of issues affect the validity of our work. For most of our experiments, we used the Intel RAPL tool, which has evolved along Intel CPU generations to be known as one of the most accurate tools for modern CPU, but still adds an important overhead if we adopt a sampling at high frequency. The other fine-grained tool we used for measurements is POWERAPI. It allows to measure the energy consumption at the granularity of a process or a Cgroup by dividing the RAPL global energy over the running processes using a power model. The usage of POWERAPI adds an error margin because of the power model built over RAPL. The RAPL tool mainly measures the CPU and DRAM energy consumption. However, even running CPU/RAM intensive benchmarks would keep a degree on uncertainty concerning the hard disk and networking energy consumption. In addition, the operating system adds a layer of confusion and uncertainty.

The Intel CPU chip manufacturing process and the materials micro-heterogeneity is one of the biggest issues, as we cannot track or justify some of the energy variation between identical CPU or cores. These CPU/cores might handle frequencies and temperature differently and behave consequently. This hardware heterogeneity also makes reproduction complex and requires the usage of the same nodes on the cluster with the same OS.

3.3.8 Conclusion

In order to increase the **accuracy** in comparative experiments, we conducted an empirical study of controllable factors that may increase the energy variations on platforms with some of the latest CPUs and for several benchmarks. This study is not intended to nullify the variability of the CPU, as some of this variability is related to the chip manufacturing process and its thermal behavior. Instead, it aims to tame and mitigate this variability along controlled experiments. In this study, We investigated some previously discussed aspects on some recent CPUs, considered new factors that have not been deeply analyzed to the best of our knowledge, and constituted a set of guidelines to increase this accuracy for energy related experiments. Some of these factors, like the *C-states* usage, can reduce the energy variation up to 500 % at low workloads, while choosing the wrong cores/PU strategy can cause up to 30× more variability.

3.4 Summary

there As seen in Section section 2.2 of the state of the art, a successful benchmark faces three challenges: reproducibility, accuracy and representativeness. This chapter has covered two of the three criteria.

The first section covered the reproducibility challenge by studying the existing techniques that ensure reproducibility. We have observed that there are two primary methods for encapsulating experimental systems. The first is to utilize virtual machines,while the second one is based on containers. We established that Docker is more suitable for energy-related studies for three reasons.

1. It is more lightweight than virtual machines.
2. It offers the interactivity with the hardware of the host machine which will enable us to gather more metrics.
3. It has a constant overhead, a key factor to nullify the encapsulation impact on the energy consumption when performing empirical analyses.

After settling on the most suitable choice to encapsulate experiments, we highlighted the need to enhance the reproducibility of empirical tests along several axes (benchmarks, metrics, and candidates) for the purpose To keep up with the rapid pace of software development. Furthermore, we have provided a model that enables to extend comparative expriment. The foundation of this model is the separation of the experiment into multiple independent components: an orchestrator that executes the experiment, one or more observers that collect metrics, the candidates being tested, and the benchmark against which these candidates are being compared.

As for the second section, we addressed the accuracy challenge in the context of energy-based experiments. We started by analyzing the impact of the chosen encapsulation method from the previous section on this challenge. to find out that the impact is negligible. Then, we conducted an empirical study utilizing some of the well-known benchmarks in literature, `Stress-ng` [8] and `NAS Parallel Benchmark` [7], on a variety of machines with diverse hardware combinations and operating system setups and tunings. We've shown that optimizing the operating system can have a big effect on how accurately energy is used. This effect can affect the accuracy of the experiments from an increase of $5\times$ to a penalty of $30\times$.

We have also shown in this section the harm that this taming of the variation can cause to the representativeness of the results,since some aspects that help increase the accuracy of the

[8]https://kernel.ubuntu.com/~cking/stress-ng

results in the research environment can't be applied in the production environment, such as turning off the C-states. Another impact of increasing this accuracy was the increase in the overall energy consumption of the tests. Even if the overall was for all the candidates, this remains an issue to be addressed.

Overall, the goal of this chapter is to establish a reproducible and accurate protocol for conducting energy-related experiments. From now on, we shall use this protocol in our studies for the purpose of reducing the energy usage of software.

Chapter 4

The Energy Footprint of Programming Frameworks

4.1 Introduction

In this chapter, we study how the programming framework affects the energy the software consumes. We suggest starting with general micro-benchmarking and watching how each programming framework performs with the CPU and memory. The main goal of this chapter is to advise developers on how to choose a programming framework based on their project's needs to make their product use the least amount of energy possible. For such a question, no answer is obvious. Nonetheless, there are some features we can take from each programming framework, such as:

- performance,
- community support,
- scalability,
- energy consumption,
- memory usage.

As we saw in the previous chapter, one of the most important things about a benchmark is how well it REFLECTS the production environment. Therefore, we extend our study to cover real-world use cases, including two case studies reported in the following sections.

4.2 Investigating Remote Procedure Call Frameworks

4.2.1 Introduction

With the success of Internet and the emergence of cloud technologies, many communication protocols compete to take the lead. In particular, most of the software architectures are now based on multi-services and micro-service technologies. And, to cope with the higher versatility of developers, multiple companies choose to open their services to different programming languages. Interestingly, this approach intends to take advantage of each programming language to satisfy specific requirements. However, the challenge nowadays is to make the bridge between those platforms. We have many initiatives, such as OpenAPI that try to create a taxonomy for RESTful APIs, while other approaches implement all the different interfaces of the protocol by themselves, such as RPC.

4.2.2 Research Questions

In this section, we first explore the ease of implementation of this protocol, and then we will try to answer the following research questions:

RQ 1: How do RPC implementations consume with regard to the size of the incoming request?

RQ 2: How do RPC implementations consume with regard to the number of concurrent clients?

4.2.3 Experimental Protocol

Measurement Context

Hardware settings. All the experiments are run on the cluster paravance of the G5K platform. This cluster is composed of 72 identical machines, each one is equipped with 2 Intel Xeon E5-2630 V3, with 128 GB of RAM. For more accuracy, our SUT (*System Under Test*) runs a minimal version of Debian 9 (4.9.0 kernel version), which enforces the core processes required for our experiment. Furthermore, we used Docker containers technology for reproducibility of the experiments and the isolation of the servers.

Client & server environments. To limit the impact of the network on the experiments, we run both the client and the server on the same machine. However, we isolate each part on a separate CPU socket to reduce the noise that the client might have on the server and

vice-versa. To do so, for each iteration, we always run the same client on socket 0 and the server that we want to test on socket 1. Both the server and the client use the whole socket for their experiment. In addition, all the additional services, such as the kernel and HwPC sensor, are run on socket 0. Therefore, the only process being executed in socket 1 is the server that we benchmark and monitor.

Key Performance Metrics

Energy measurements. To report on the energy consumption, we used HwPC sensor [TOCITE], which is based on Intel RAPL technology, one of the most accurate tools to measure the energy consumption of the CPU and DRAM [TOCITE]. For better accuracy, we ran the HwPC sensor with a frequency of 1 Hz, and we used the same machine for all the experiments to reduce the variability [TOCITE].

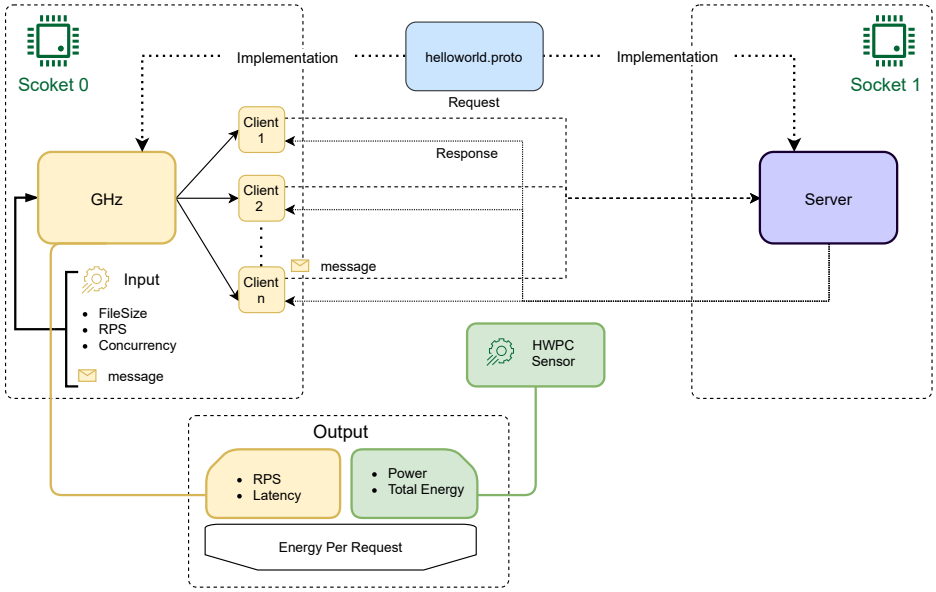


Figure 4.1: Experimental software architecture.

Performances. For better accuracy and more details, We use an updated version of the open source RPC benchmarking tool, named GHZ (https://ghz.sh/). The modified version allows us to monitor the average power for each request from both the server and the client sides. The new version is available in the repository.[1]

[1]https://github.com/chakib-belgaid/energy_ghz

Input Workload

The purpose of the experiment is to analyze the behavior of different GRPC implementations. Therefore, we have two kind of worlkoads: *number of clients* is the number of concurrent clients that we want to benchmark (handled by the GHZ client) and the *payload* reflects the size of each request (varies from 50 Bytes up to 10 MB). The client consumes the protocol description found in the file `helloworld.proto` to generate an implementation for the message and then forks multiple instances that send the same request to the server.

Candidates

The server implementations are based on the official implementation by Google for most of the languages. Each server uses 16 cores and is limited to 512 MB of RAM. Each implementation is packaged as a Docker image, which will allow us to add new implementations easily.

Extension

For the sake of experimental extensibility, we provide a GitHub repository that contains the implementation of all our experiments.[2] Adding a new RPC **candidate** can be achieved by creating a new Docker image and putting it in a new folder named after the language. As for the **workload**, it can be extended easily by adding new files in the folder **payload**.

4.2.4 Results & Findings

RQ 1: How do RPC implementations consume with regard to the size of the incoming request? The purpose of this question is to study the energy consumption of the RPC server when transferring large objects. To do so, we send $80,000$ requests to the server whose size scales from 10 bytes up to 10 Megabytes, resulting in $10,000$ requests per size per server. To eliminate extra factors, we let the server handle the rate at which it can answer each request. However, we put a 20 seconds timeout deadline for each request. Therefore, our boundary condition is only the number of requests received by the server. For this experiment, we investigate 4 observable variables:

1. the average power consumption during the scenario indicates the overall behavior of the server when working for long durations,

2. the tail latency for the 99th percentile, which indicates how efficient is the server,

[2] https://github.com/chakib-belgaid/energy_benchmarking_grpc2

3. the average number of requests per second, which indicates the average number of clients that the server can handle,

4. the average energy cost of a single request: unlike the first indicator, this one shows how green is the implementation considering performance.

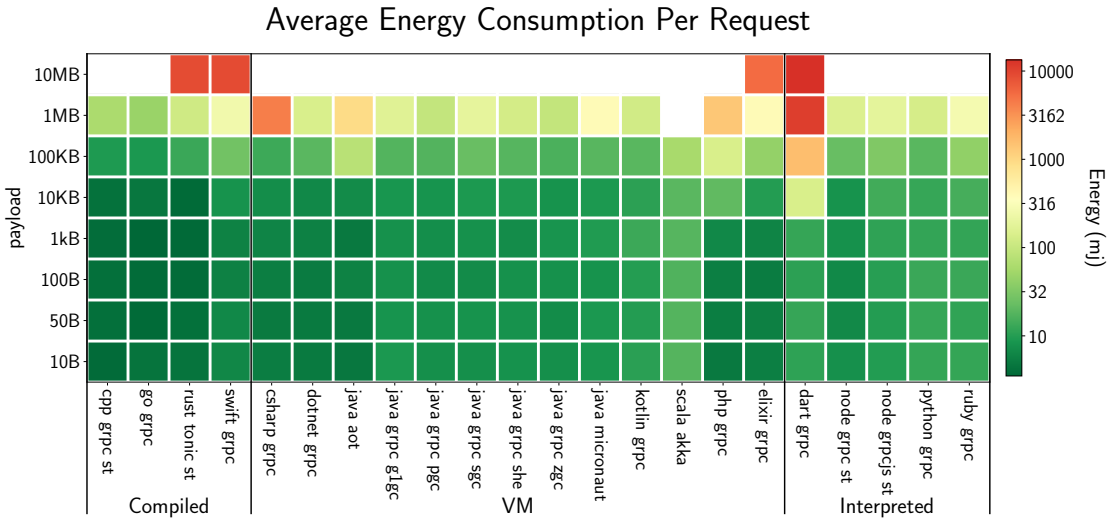


Figure 4.2: Energy consumption based on the request size

Figure 4.2 depicts the overall behavior of each framework based on the size of request (payload). For each framework, we can distinguish three modes, and they all depend on the payload size:

1. Stress free mode when the server has enough resources to satisfy the requests, as they require a memory less than a certain threshold (depends on the language and the platform),

2. Escalation mode when the requests tend to be bigger, but the server can still manage to handle them, at the price of a change in the energy and performance behaviors,

3. Broken state mode when the requests are much heavier and the server break—like 10 MB.

Stress Free Mode

In this mode, the compiled languages tend to consume fewer resources (average power). JVM-based languages tend to consume more energy, especially Scala. However, we do not observe the same behavior when it comes to efficiency. Unlike the other interpreted programming languages, PHP performances could be compared to the compiled ones, such as CPP or GO, and even better to some others, such as Swift. JVM-based languages tend to have better performances than the interpreted ones. Furthermore, OpenJDK has shown more efficiency than GraalVM []. Overall, we can have 3 groups when it comes the cost of each request:

- Energy-efficient class: C++, GO, RUST, ELIXIR, and PHP,
- Middle class: Most of the interpreted languages and VM-based ones,
- Energy-greedy class: Crystal and Scala.

Escalation Mode

In this mode, the behavior of the server depends on the payload. We observe three cases:

1. Drop in performances without an increased power, such as .Net core, Java micronaut, Crystal, and Dart. In this case, the server keeps using the same resources, and sometimes less, because it takes more time to handle the fewer requests. This class of languages tends to be the most energy-consuming when it comes to the cost per request;
2. Increase in power without affecting the performances, such as Go or .Net. The energy consumption of a single request, is affected slightly but still increases;
3. Increase in power and drop in performances. Despite the increase of power consumption, the server becomes slightly slower, which increases the energy cost per request. This cost is still better than the first case, which concludes that the servers in the first category are on the verge of breaking.

We can mention the case of Elixir that keeps scaling despite the lack of performances compared to other compiled languages (Go, CPP).

Broken state mode

Only four of the 25 configurations could parse the 10 MB files, and only 1 from those could achieve a 76% acceptance rate which is Elixir, the other 3 had less than 3% success rate (Rust, Swift and Dart). The rest could be divided into two categories:

- Timeout where requests took too much time that the client canceled them, in this category we find most of dynamic codes, such as OpenJDK and Kotlin,
- Size of request exceeded the maximum size when the implementation could not handle requests with large size, as observed with .Net, Go, .Net core, CPP, PHP, Scala, Nodejs, Ruby, Python.

[RQ 2:] How do RPC implementations react to the number of clients ?

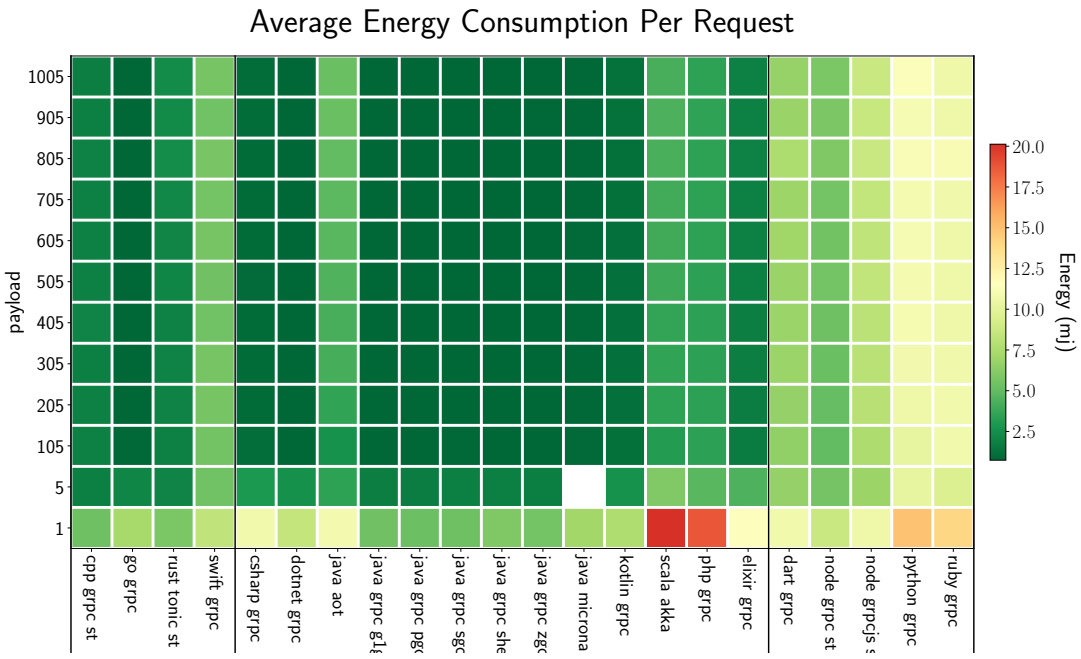


Figure 4.3: Energy behaviour based on the number of the clients

Power behaviour

Based on the heatmap, we can distinguish two modes:

- Low number of clients when the number of concurrent clients is below 100,
- Moderate to high number of clients when the number of clients exceeds 100.

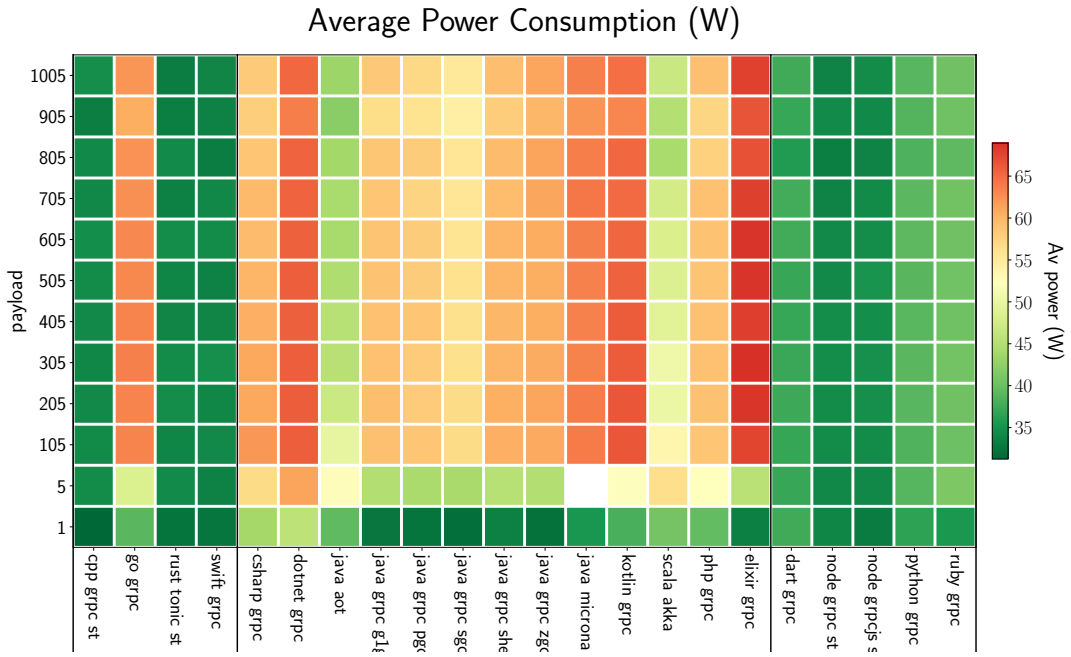


Figure 4.4: Average power consumption based on the number of the clients

Lite mode The benchmarked implementations can be grouped into two classes:

1. Energy-efficient frameworks where most of the framework's power consumption is around 33 Watts.

2. Energy-greedy frameworks where the average power consumption is higher than 37 Watts.

In each programming category, we observe both energy-efficient and energy-greedy behaviours. Therefore, we conclude that it depends more on the implementation of the library itself, rather than the category of the programming language. Scala and Kotlin are an excellent example to support this hypothesis, as both of them run on the same virtual machine as Java (OpenJDK 16.1). Yet, their average power is 130% higher than the Java implementation.

Stressed mode Although the same classes remained the same, not all the languages had the same evolution and here we can clearly observe a correlation with the category of the programming language rather than the implementation itself. We can clearly highlight that VM-based languages have a significant increase (double) in the average power consumption

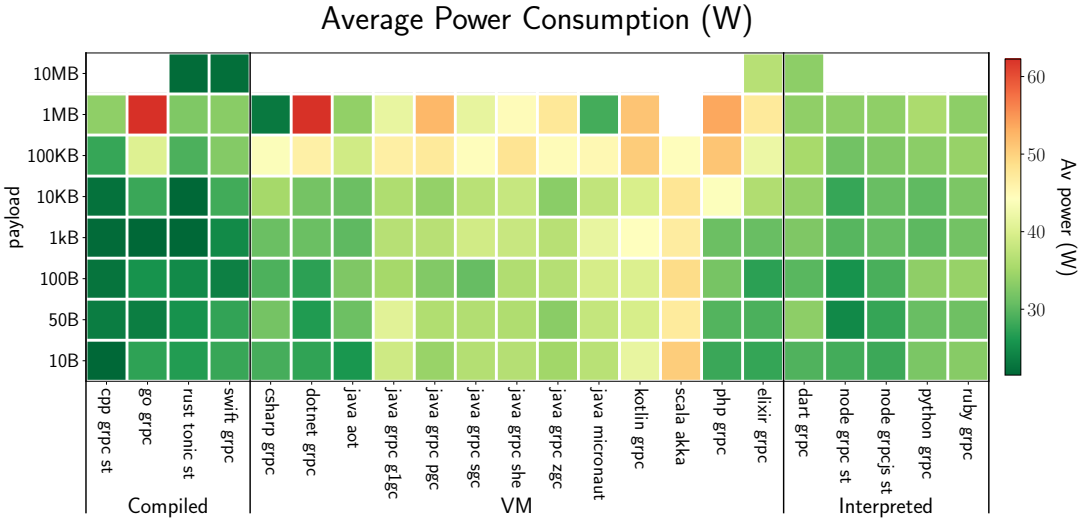


Figure 4.5: Average power consumption based on the request size

after they receive more than 100 concurrent clients. Except PHP, all the interpreted and compiled languages preserved their energetic behaviour. Our hypothesis points to the JIT, since it compiles the code and makes it run faster, hence stressing the CPU. A interesting behaviour has been noticed for the GraalVM: the decrease of energy consumption when increasing the number of the clients. This is related to the drop of the performances, which was probably due to the bottleneck situation where the GraalVM could not handle more than 100 clients simultaneously.

Performance Behaviour

In this section, we study only the number of requests per seconds processed by the server without looking at its energy. We consider three observable variables:

- Satisfaction ratio: how many requests have been satisfied among the total requests,
- Request Per Seconds: The number of the requests that have been answered from the server,
- Tail Latency at 99%: one of the best metrics to evaluate the performances of a server.

Satisfaction ratio Most of the considered frameworks satisfy all the requests, by either reducing the number of requests per second or by increasing the processing time. However, there are some frameworks that have chosen a different approach, such as Dart or Scala, where the choice was to keep a certain limit of latency even if not all the requests are answered. Furthermore, we tend to observe this behaviour among other frameworks, such as Python or Asynchronous NodeJs, when the number of the client exceeds 800.

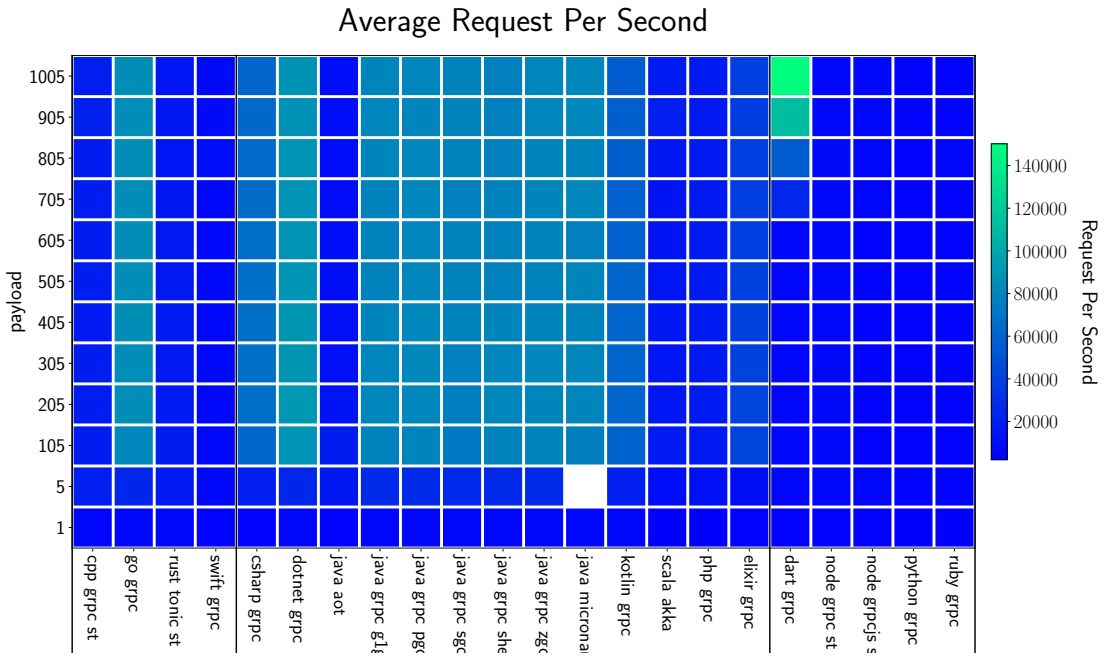


Figure 4.6: Number of requests per second based on the number of clients

RPS Most of the servers hit their RPS limit after 5 clients and 100 clients for vm based servers, and after this the number keeps constant, which will decrease the average RPS per client. .Net server is the most performant, followed by Java and Go, while Python and Ruby are the least performant.

Tail Latency The increase in the number of requests per second, does not necessarily mean a lower latency. As one can notice in Fgirue 4.8, until the 1000 clients, Go provides the least latency beside .Net. GraalVM provides the highest latency, on average. However, Dart tends to become slower when we increase the number of clients, until we pass the 600 simultaneous clients, and there it changes its behaviour, instead of satisfying most the

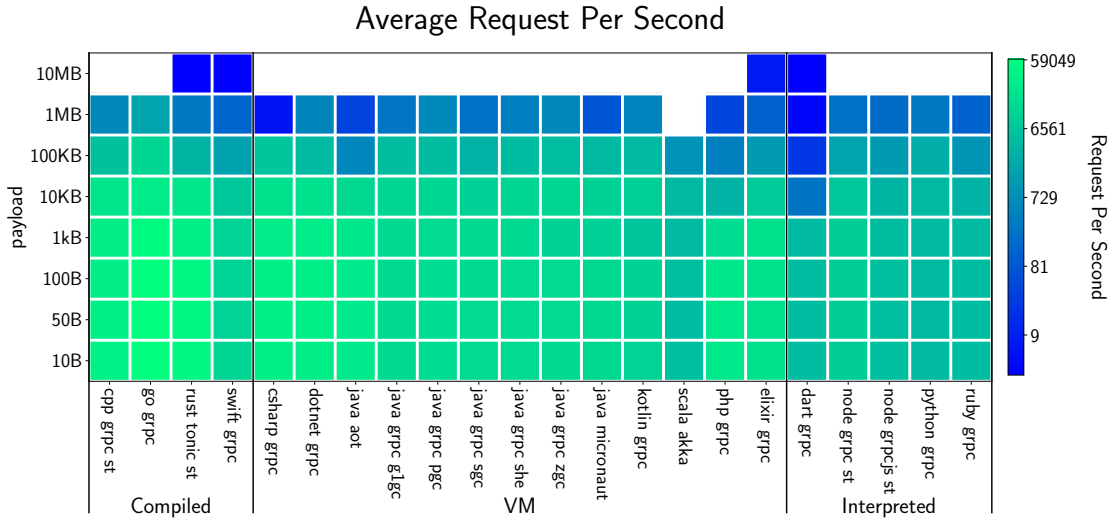


Figure 4.7: Number of requests per second based on the request size

requests it notify the clients directly that the server is saturated, hence a drop in satisfaction ratio, and an amelioration for the average latency.

Energy Per Request

Now, after we made a separation between the energy and the performances, we have seen that most of the performance servers tend to be energy-greedy, so we propose to investigate this trade-off between energy and performances. To do so, we report on an average cost of a single request, in Joules. Except GraalVM when the cost of the a single request increases with when we add more clients, all the frameworks report on a constant cost, Java, .net and go are the most energy efficient, while Python and Ruby may cost up to 10x more. Therefore, we conclude that the number of clients does not impact the energy significantly. Then, we study how the payload size of the requests impact the energy consumption of the framework.

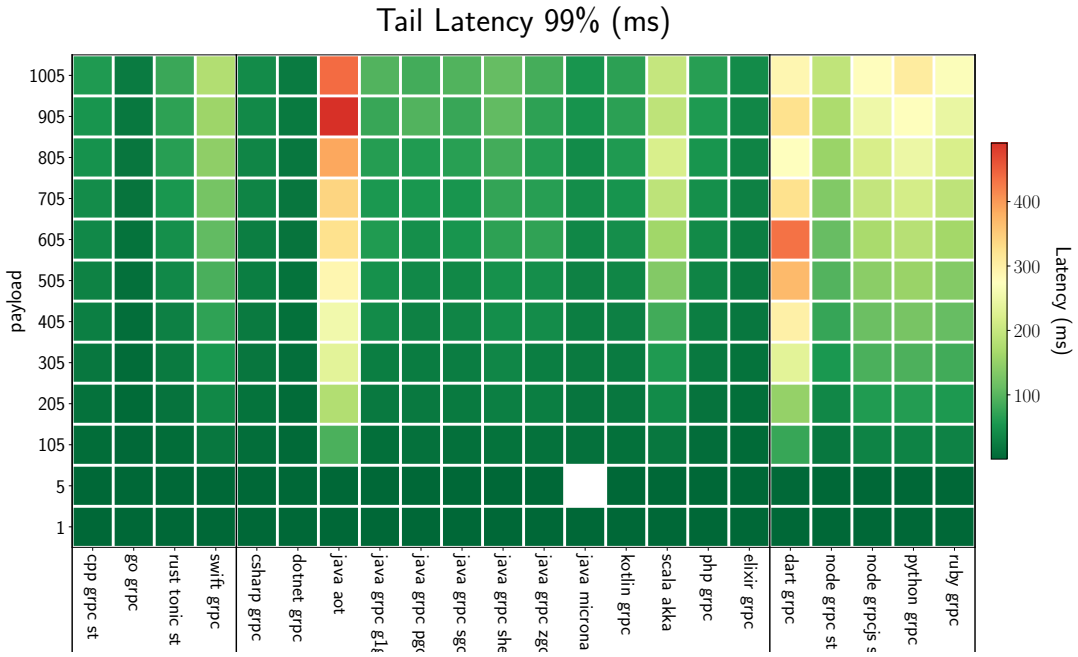


Figure 4.8: Tail latency (99%) based on the number of clients

4.2.5 Threads to Validity *TODO : missing*

4.2.6 Conclusion *TODO : missing*

4.3 Investigating Web Application Frameworks

4.3.1 Introduction

Nowadays, web applications are dominating online systems. From Google to Facebook and others, web applications are widely deployed across organizations and continuously accessed by end-users, both for their personal and professional daily tasks. In practice, the development of these web applications heavily relies on a wide ecosystem of *web frameworks*, which are intended to ease and foster the development process. However, once deployed, the applications developed with such web frameworks do not exhibit the same performances, as reported by the *Web Framework Benchmarks* periodically published by the TECHEMPOWER company.[3] Thanks to such benchmarks, developers can take informed decisions on the most efficient technology to adopt to implement their web applications. Unfortunately,

[3]https://www.techempower.com/benchmarks

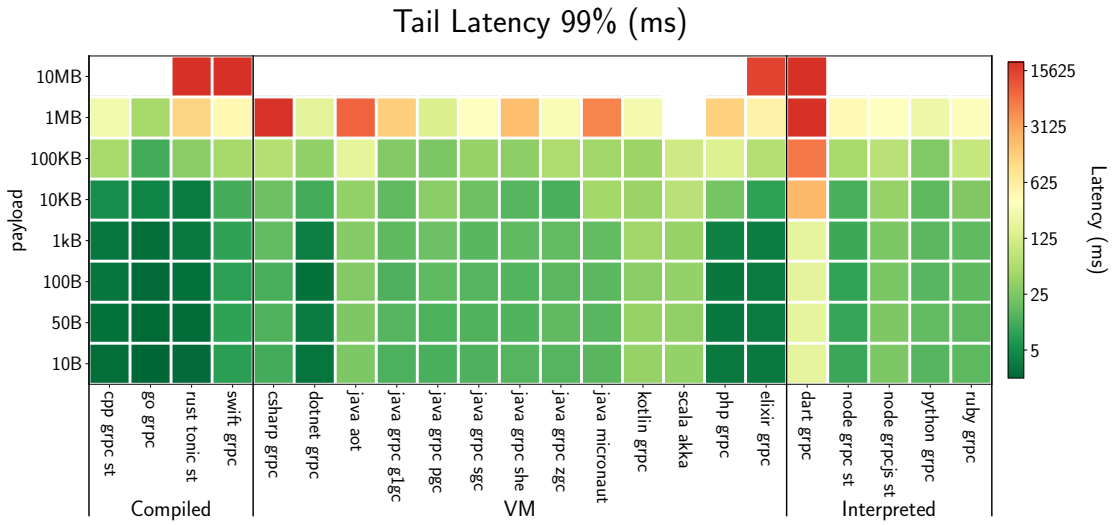


Figure 4.9: Tail latency (99%) based on the request size

one can regret that developers and benchmark providers mostly focus on popularity and performance criteria when picking a web framework, with fewer considerations for the resource consumption implications of their choice. This is all the more regrettable that cloud providers are more and more adopted by developers to host these web applications. While cloud providers offer a convenient elastic provision of resources to scale according to application requirements, this convenience may induce critical costs for their business.

Beyond the economical cost of web applications, one can also question the global impact of web applications on worldwide carbon emissions. Given the tremendous success of web applications, their deployment has severely increased over the last years, thus causing a rebound effect on the power consumption of server infrastructure—being hosted or supported by cloud providers. While one can challenge the relevance of features that are continuously deployed by developers to keep engaging end-users, reconciling economical and environmental concerns remains an open challenge to address.

Given this context, this section intends to address this challenge by investigating the energy footprint of web frameworks. In particular, we aim to support the developers of web applications with relevant guidelines that can help them to choose the web framework that is not only the most popular or provides the best performance but also exhibits a low energy footprint. By minimizing the energy consumed to process user requests, with no service

quality penalty, developers can reduce the operational cost of their web applications and contribute to reducing worldwide carbon emissions of ICT.

To achieve this objective, we leverage the TECHEMPOWER *Web Framework Benchmarks* to incorporate server-side energy measurements obtained from a software-defined power meter, named POWERAPI [37]. These measurements are then analyzed in depth to understand the key criteria that can impact the power consumption of web frameworks and derive guidelines to can support developers to pick the most energy-efficient web frameworks according to their requirements.

Comparison of Web Frameworks

Many studies have been conducted to compare the performance of web frameworks. One can cite [39] who compared two of the most famous Java frameworks—Play and Spring—or the work of [10] who compared different PHP frameworks using 6 criteria: intrinsic durability, industrialized solution, technical adaptability, strategy, technical architecture, and speed. In our context, we push a 7^{th} criterion that impacts the economic outcome of the project.

Energy Efficiency in Software Engineering

In their paper [92], the authors studied the impact of programming languages on energy, time, and memory by using the CLBG benchmark, where they executed 10 different benchmarks[4] across 27 wellknown programming languages [2]. The work of these authors was an extension of a research initiated by the work of Couto et al. [29] to measure the impact of programming languages choice in real-life applications, instead of micro-benchmarks.

Irene *et al.* [74] investigated the impact of web servers on energy when handling web applications. They analyzed 7 applications executed within 4 servers in 38 different scenarios. The authors showed that the energy greatly depends on the web server, but the impact of the application may also influence the energy behavior of the server. In their approach, they used measured the energy consumption during the integration tests, while we are interested in simulating more realistic workloads, and isolating the energy consumption of the server from the client's one.

Other works have been achieved on measuring the energy consumption of the client applications, as an example [93] concluded that there is a variation among the different websites and the impact of the browser on this energy consumption.

[4]https://salsa.debian.org/benchmarksgame-team/benchmarksgame

4.3.2 Research questions *TODO : missing*

4.3.3 Experimental Protocol

In this section, we describe the environment we used during the experiments, covering hardware features, experiments of the framework and the methodology.

Measurement Context

The purpose of this experiment is to highlight the energy impact of the technology stack used to develop a web application once in production.

Candidate Frameworks Overall, we selected 210 web frameworks to be evaluated in this study. Each framework may have multiple configurations, based on the database, alternative interpreters, etc. Table **??** lists the frameworks that we selected for this study.

Table 4.1 highlights the number of frameworks used in the experiment per category of benchmark. As we see in this table, some of the frameworks worked on certain conditions, while they failed on other benchmarks, such as Nickel (based on Rust). While it might be one of the most energy-efficient Rust frameworks, Nickel does not work with databases. Therefore, it cannot be used for any situation, but if a (stateless) web application does not interact with a database, then it might be the best choice. Many reasons are behind the observed failures, either there was no implementation or some errors were raised when handling the request.

Remark We decided to skip the idle part in the validation benchmark since it is not relevant.
TODO: ROMAIN: Something is missing here

- orchestrator is responsible for creating Docker images, selecting and launching the benchmarks,
- web server, or the *system-under-test* (SUT), is the machine responsible for launching the framework by mean of the pre-installed power meter,
- database server offers the database that will be used by all the frameworks during the benchmarks.
- client machines avoid the bottleneck on the client's side, client requests are sent from another machine (one or many) that simulates hundreds of concurrent connections to the framework,

Table 4.1: Number of available web frameworks per programming language.

Language	Bb	Query	Update	Plaintext	Fortune	Json	Total
c	1	1	1	6	1	5	15
c#	21	20	14	12	14	17	98
c++	27	16	14	20	13	25	115
cfml	2	1	1	1	1	2	8
clojure	8	8	5	6	7	8	42
common lisp	2	/	/	/	/	2	4
crystal	3	1	/	2	/	2	8
d	3	2	1	2	1	3	12
dart	/	/	/	2	/	2	4
elixir	1	1	/	/	/	1	3
erlang	3	2	/	3	1	3	12
f#	/	/	/	4	2	8	14
go	19	18	16	15	15	19	102
groovy	1	/	/	1	/	2	4
haskell	1	1	1	2	1	2	8
java	20	20	18	26	21	26	131
javascript	19	19	16	14	17	14	99
julia	/	/	/	1	/	1	2
kotlin	10	9	6	5	5	10	45
lua	1	1	/	1	1	2	6
nim	/	/	/	2	/	3	5
ocaml	4	4	3	1	2	5	19
perl	2	/	/	1	/	2	5
php	22	18	15	10	12	14	91
prolog	/	/	/	1	/	1	2
python	31	21	15	17	16	30	130
racket	1	/	/	/	/	/	1
ruby	23	15	11	8	12	19	88
rust	8	7	6	9	8	10	48
scala	7	6	3	8	5	11	40
swift	2	2	/	2	/	2	8
typescript	4	2	2	3	2	6	19
v	/	/	/	1	/	1	2
vala	/	/	/	1	/	2	3
vb	2	2	2	1	2	1	10
Total	**248**	**197**	**150**	**188**	**159**	**261**	**1,203**

- recorder collects the power measurements from the SUT and the key performance metrics collected by the clients ***TODO: add link to the measurement process***.

The tests have been executed in machines from the cluster chetemi [5] of the grid5000 **??** platform.

TODO: add hardware description

Note It has been proven in the work of Eddie Antonio Santos et al. that Docker does not impact the energy consumption. Thus, using containers and isolation avoids any noise of the operating system after executing one benchmark and contributes to the reproducibility of our results.

Input Workload

To compare the energy consumption and performance efficiency between multiple frameworks, each framework is used to implement the same web application—*i.e.*, replying to the same HTTP endpoints and requesting the same database. Then, we run the same sequence for all the SUT:

1. lunch the web application,
2. wait for 20s for the warmup,
3. measure the average power when the application is in idle state,
4. using multiple clients, we send the same request concurrently during 20s,
5. increase the number of parallel requests,
6. measure the energy during this execution,
7. change the request type,
8. repeat from the 3^{rd} step.

The following sections describe each type of experiment and the purpose behind it, by giving some examples of the expected responses.

Test Scenarios We have 7 categories of benchmarks:

[5] https://www.grid5000.fr/w/Hardware

Idle In this benchmark, we measure the idle energy consumption of the web framework: this reflects the average energy consumption of an application during periods without connections. For example, a company website beyond working hours or a online shop at night.

Single Query During this benchmark, each request is processed by fetching a single row from a simple database table. This row is then serialized as a JSON response, then returned to the client. This is the most common type of request in a web application. For this benchmark, we use a variable number of clients to measure the energy consumption of the web framework when it is under load.

Multiple Queries This benchmark aims to observe the behavior of a web framework when it processes multiple entries from the database. Therefore, each request is processed by fetching multiple rows from a simple database table and serializing these rows as a JSON response. In this case, we use a 512512 clients.

Fortunes In this benchmark, the framework's ORM is used to fetch all rows from a database table containing an unknown number of Unix fortune cookie messages (the table has 12 rows, but the code cannot have foreknowledge of the table's size). An additional fortune cookie message is inserted into the list at runtime and then the list is sorted by the message text. Finally, the list is delivered to the client using a server-side HTML template. The message text must be considered untrusted and properly escaped and the UTF-8 fortune messages must be rendered properly.

Update Queries This benchmark exercises database writes. Each request is processed by fetching multiple rows from a simple database table, converting the rows to in-memory objects, modifying one attribute of each object in memory, updating each associated row in the database individually, and then serializing the list of objects as a JSON response. The maximum number of clients is 512512. The response is analogous to the multiple-query benchmark.

Plain Text In this benchmark, the framework responds with the simplest response: a `"Hello, World"` message rendered as plain text. The size of the response is kept small so that gigabit Ethernet is not the limiting factor for all implementations. HTTP pipelining is enabled and higher client-side concurrency levels are used for this benchmark.

JSON Serialization In this benchmark, each response is a JSON serialization of a freshly-instantiated object that maps the key message to the value `"Hello, World!"`. For each one of the above scenarios, we consider different levels of stress and Table 4.2 shows the different levels for each scenario.

Table 4.2: Stress levels for each scenario.

Scenario	type of stress	level 1	level 2	level 3	level 4	level 5	level 6	level 7
Single Query	Number of parallel clients	16	32	64	128	256	512	/
Multiple Queries	Number of rows to read from the database	1	5	10	15	20	30	50
Update Queries	Number of rows to update in the database	1	5	10	15	20	30	50
Fortunes	Number of parallel clients	16	32	64	128	256	512	/
JSON Serialization	Number of parallel clients	16	32	64	128	256	512	/
Plain Text	Number of parallel clients	256	1024	4096	16384	32384	/	/

To include additional scenarios, one might implement a Python class that handles the metadata of the workload, such as the query route, the query parameters, and the expected results.

Key Performance Metrics

We focus on comparing the energy behavior of different frameworks in multiple scenarios. To measure the energy consumption of those frameworks, we launch each one for a fixed duration, then all the clients send multiple requests simultaneously. We compute the number of satisfied responses, which reflects the performance of the framework, the average latency and the global energy consumed during the whole period to deduce the energy cost of each request.

Remark : Before each benchmark we consider a warmup period of 10 seconds to let the framework to reach its steady state.

Runtime Measurements

- energy measurement : we use PowerAPI [15], a software power meter to gather the power consumption of the SUT, after we project the timestamps of each experiment phase to calculate the energy consumption of the framework during this phase. Energy is an integral of power over time, so we use a numerical approach to isolate the energy consumption.After this, we divide the calculated energy per the number of responses.

$$E = \int_b^a P(t)dt \simeq \sum_{k=1}^{n} \frac{P(t_k - 1) + P(t_k)}{2} \tag{4.1}$$

- Total cost of the energy during each period,
- Total number of requests,
- Average latency,
- Average energy cost per request.

Due to some technical problems, not all the web frameworks returned the tail latency (99%), therefore we substitute it with the average latency during this study. However, for further details, the reader can always check the available values in our public repository.[6]

Architecture We aim to compare the energy consumption of different web frameworks. For this, we consider the web framework as a black box and we take into account the response to the 6 previous scenarios using the same database. To isolate the energy consumption of the web framework, the benchmark is run in a separate machine with the minimum services and the power meter. Figure 4.10 illustrates the architecture of our system.

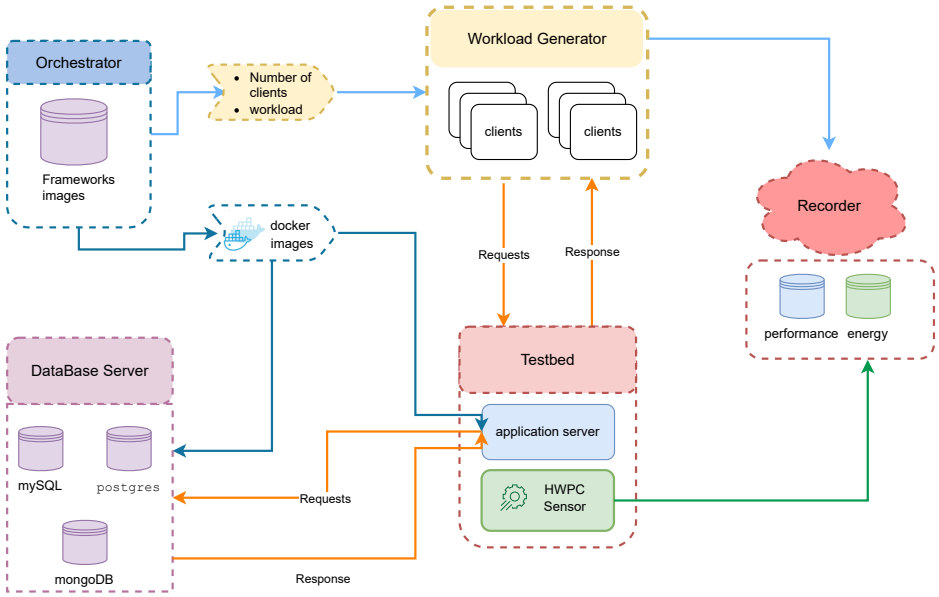


Figure 4.10: Architecture of the experiments.

Bias Analysis We are aware of potential bias analysis regarding the estimation of the total energy cost, the interference of other system processes during the execution and some external events. Thus, we run experiments multiple times and compute the average values.

[6]https://github.com/chakib-belgaid/frameworks-benchmarks-results

Extension

To follow the guidelines that we presented in Chapter 3, we provide a GitHub repository[7] where one can add extra **candidates** by creating a new project using the option `-new`. Then, interested practitioners have to fill the template and provide the Docker image file. Additionally, to configure the **workload** we provide the option `-concurrency-levels` and `-duration`. The choice of the database is included in the Docker image.

4.3.4 Results & Findings

Overall we had $8,750$ benchmarks, and all can be found in the online repository.[8] In this section, we discuss these results to answer the following questions:

- is there a dominating programming language when it comes to performance, energy consumption, and latency?
- which class of programming language is performing well?
- is there a correlation between energy consumption and latency?
- is there an impact on the server when it comes to changing the database?

Since most of the companies use the same stack, we do not aggregate the results as it may lead to confusion. For example, comparing the average energy consumption of each programming language will not reflect reality, particularly when we have two web frameworks at the opposite ends of the spectrum.

Overall Statistics

To determine which web framework/stack is performing well, we need to establish some general idea about the average energy consumption and latency of the frameworks under study. Instead of reporting the raw energy consumption of those frameworks, we will provide some green factors to determine which one is eco-friendly and which one is greedy. In this part, we will discuss the average behavior of the frameworks, highlight some trends, and eliminate the outliers. As we said in the threats to validity, being an outlier in this case does not mean that the web framework is not performing well, it means that the web framework is not performing well in the same way as the others within the context of this experiment.

[7] https://github.com/chakib-belgaid/FrameworkBenchmarks
[8] https://github.com/chakib-belgaid/frameworks-benchmarks-results

On the other hand, the cost of a single request is proportional to the number of requests per second of the web framework. To narrow the research space, we will look for a correlation between the metrics. The Pearson correlation coefficient [114] will be used to determine this correlation. Because the Shapiro-Wilk [100] test yielded a p-value of 0.0 for all metrics.

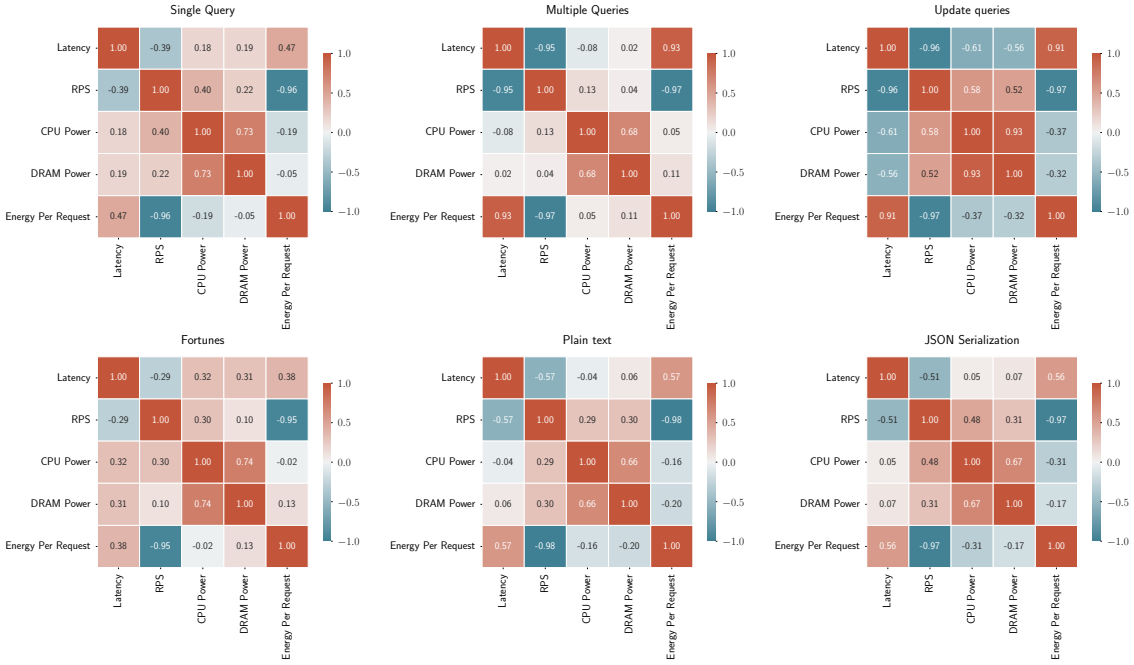


Figure 4.11: Spearman Rank Correlation between different metrics

Figure 4.11 depicts the correlation between the metrics for the 6 scenarios. The Pearson correlation coefficient quantifies the linear correlation between two variables X and Y. It ranges from -1 to +1, with 1 being total positive linear correlation, 0 representing no linear correlation, and -1 representing whole negative linear correlation. The stronger the correlation, the closer the value is to 1 or -1. The weaker the correlation, the closer the value is to zero. The Pearson product-moment correlation coefficient is another name for the correlation coefficient. This correlation coefficient is also known as the Pearson product-moment correlation coefficient. One can notice that there is a strong correlation between the energy consumption of the CPU and the DRAM for most of the scenarios. Moreover the average energy consumption of DRAM is one sixth of the CPU energy consumption. Therefore, in this study we will focus more on the CPU energy consumption. For more insights about the DRAM consumption we refer the reader to our github repository.[9]

Another strong correlation is between the number of requests per second and the average cost of a single request. This is because the cost a single request is proportionate to the

[9]https://github.com/chakib-belgaid/frameworks-benchmarks-results

number of requests per second of the framework, since the Average Power consumption remains constant after a certain threshold of clients.

Unlike the *multiple queries and update* queries, another scenario depicts a weak correlation between latency and number of requests per second. The reason behind such an anomaly is the fact that we summarized the data when we had a multiple number of clients. If we calculate the correlation when we have a fixed number of client, like the case of update queries (512 clients), one can notice a strong correlation.

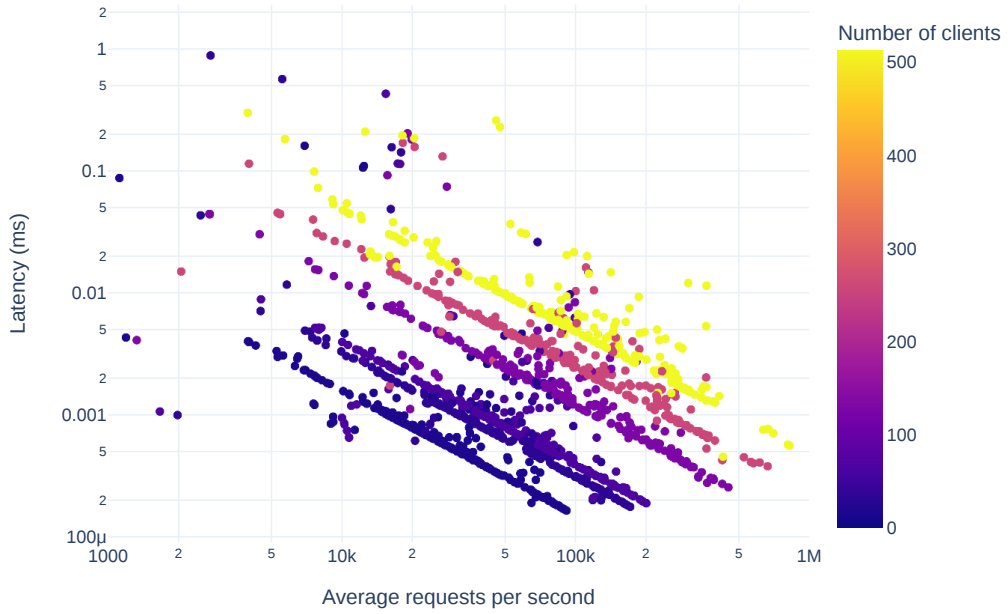


Figure 4.12: Correlation of latency and number of requests per second for a single query

Figure 4.12 demonstrates such a behavior. As one can notice, for each level we observe a linear clustering. Therefore, we can safely focus our analysis on two variables, *number of requests per second* and *average energy consumption*. The first one will indicate the performance of the solution meanwhile the second one will be used to measure how green a framework is.

Scenario-based Analysis

This section will focus on the behavior of all the implementations for each scenario. For visibility purposes, we will group the frameworks not by language but by family, so we will have 5 families:

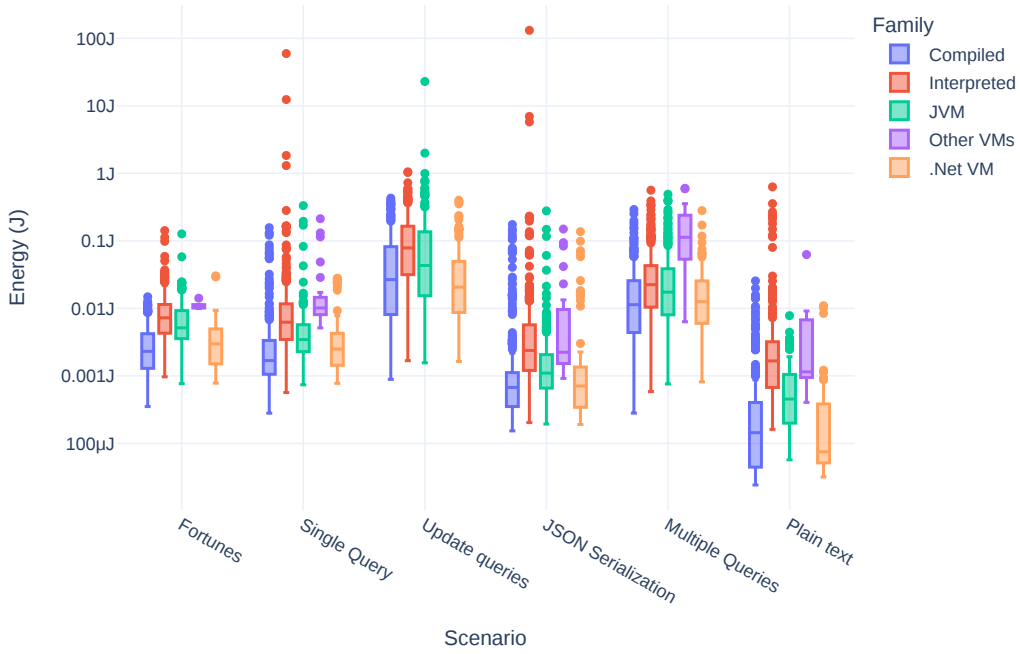


Figure 4.13: Energy consumption per request for each family of programming languages

- **compiled languages**: Rust, C, GO and C++;
- **interpreted languages**: Python, PHP and Javascript;
- **JVM-based languages**: Java, Kotlin, Scala and Clojure;
- **.Net-based languages**: C#, F# and VB;
- **Other VM-based languages**: Dart and Elixir.

Figure 4.13 depicts the programming language and the family for each framework.

Idle behviour This part will treat average power behavior when the framework is in a rest mode. Figure 4.14 presents, a density plot for each family.

As one can notice at rest, most of the families consumes between 20 and 40 Watts, 6% of the compiled languages frameworks consumes less than 15 Watts. However, 50% of Java solutions tend to consume around 50 Watts, which makes it the most greedy family. If we look at the each of the programming languages from Java, separately in Figure 4.15, we find that Java-based implementations tend to consume around 50 Watts, while Kotlin, Clojure and Scala consume around 30 Watts.

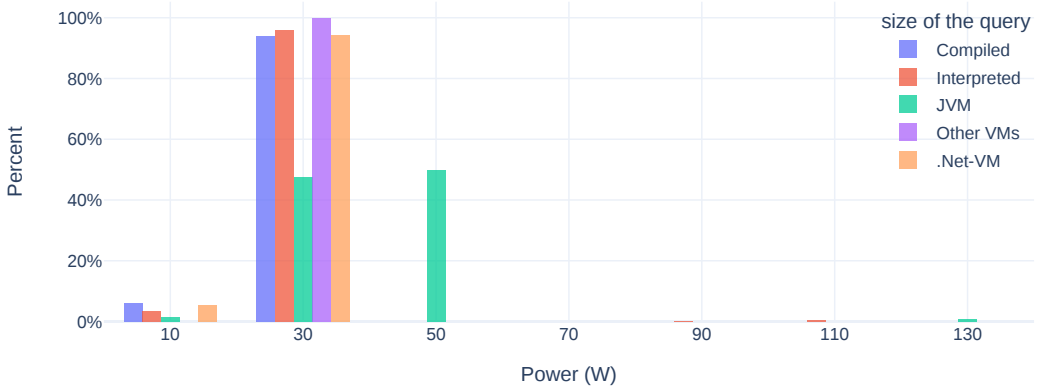


Figure 4.14: Average power consumption for the idle scenario

Single query As mentioned before, the purpose of this scenario is to benchmark the framework efficiency to handle a single entry. To determine the general behavior of the frameworks, first we will start with an histogram of average power consumption. Figure 4.16 reports on the density plot of average power consumptions for all the experiments depending on the number of concurrent clients. As one can observe, there are three main states:

1. *relaxed state* where the number of clients is less than 16: most of the frameworks consumes around 70 Watts;

2. *average state* where the number of clients is between 16 and 64: most of the frameworks consumes around 100 Watts;

3. Finally the *stress state* beyond 128 concurrent clients: most of the frameworks have a stable power consumption regardless of the number of clients. This is due to database server, which reached its maximum capacity.

Now that we have seen the overall distribution of the power within the single query scenario. We analyze each family separately. In addition to that, we include the number of *requests per second* (RPS) as a performance metrics of interest. In Figure 4.17, each run is represented by a circle, and the size of the circle represents the number of concurrent clients: The smaller the circle the less clients. On the one hand, one can notice that compiled languages are the most efficient in terms of performance, despite their low energy consumption. Moreover, there is no significant change in the average power consumption

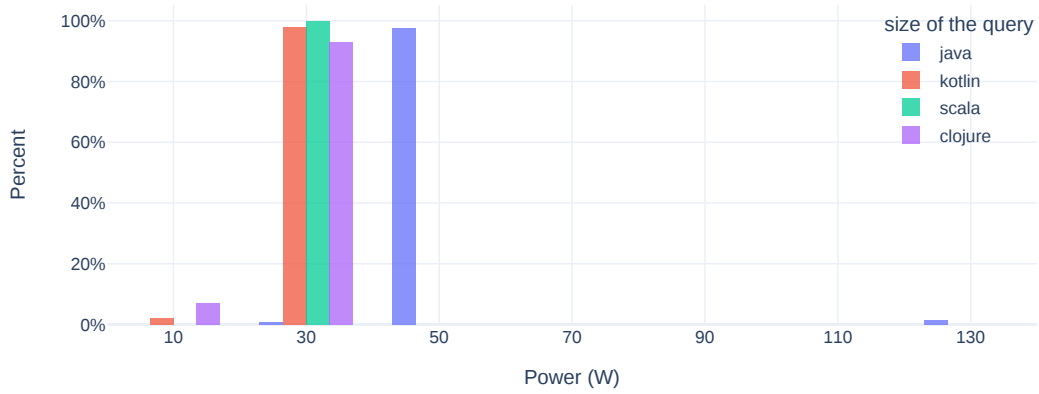


Figure 4.15: Average power consumption for Java-based languages in the idle scenario case

when we increase the number of concurrent clients. On the other hand, the JVM-based frameworks tend to consume the most energy while reporting the same performance as the .Net-based ones. Finally, the interpreted languages lack in terms of performance while keeping low average power, except for PHP, as it is has one of the highest RPS with a half million RPS which got beaten only by C++ and Rust.

Multiple queries This scenario is used to benchmark the framework efficiency to handle multiple row queries. As mentioned before, this study focuses on a fixed number of concurrent clients while we increase the size of the request per level. Figure 4.18 reports on the average power consumption for each level. As one can notice, the size of the query has no strong impact on the average power consumption. Furthermore, one can notice a slight decrease in the average power consumption (from 110 watts to 90 watts) when the size is bigger than 10 rows. This might be related to the time taken by the database to process the query. Therefore, one can conclude that the size of the query has more impact on the database than the framework itself.

Table 4.3 details the average power consumption per level for each database. One can see, that for MySQL there were no changes regardless of the size of the query, while for Postgres and MongoDB there is a slight decrease in the average power consumption when the size of the query is bigger than 10 rows.

Now that we have seen the overall distribution of power within the multiple query scenario, we can analyze each family separately. We consider the number of requests per

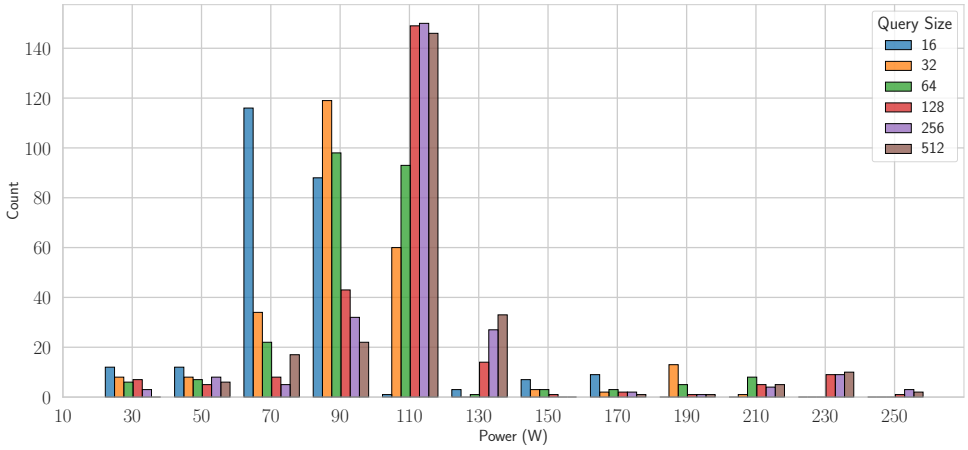


Figure 4.16: Average power consumption for the Single query test

Table 4.3: Average power consumption of frameworks based on the database type

Query size	1	5	10	15	20	30	50
MongoDB	97.17	96.93	93.38	92.58	91.61	92.585	91.17
MySQL	113.86	112.92	112.74	113.05	112.13	112.62	112.16
PostgresSQL	113.86	108.25	106.19	102.97	103.41	101.95	102.96

second (aka RPS) as related performance metrics. Figure 4.19 presents the total RPS per level for each framework in a logarithmic scale. As one can notice, the difference between the best performing framework, aka Lithium,[10] and the worst one, aka hapi-nginx,[11] is 4,000 times, while the average in power consumption is 5 times (120 for lithium vs 25 for hapi). This highlight the importance of target scale of the application when choosing the framework. Java-based frameworks tend to consume more power compared to other languages with a slight increase in performance. PHP remains one of the most efficient frameworks in terms of performance, while keeping low average power consumption.

Update This scenario benchmark the framework efficiency to handle update queries. As mentioned before, for this study we will focus on the a fixed number of parallel clients while we increase the size of the request per each level. Figure 4.20 presents the average power consumption for each level. As one can notice, the size of the query has not a strong

[10]https://matt-42.github.io/lithium/
[11]https://github.com/hapijs/hapi

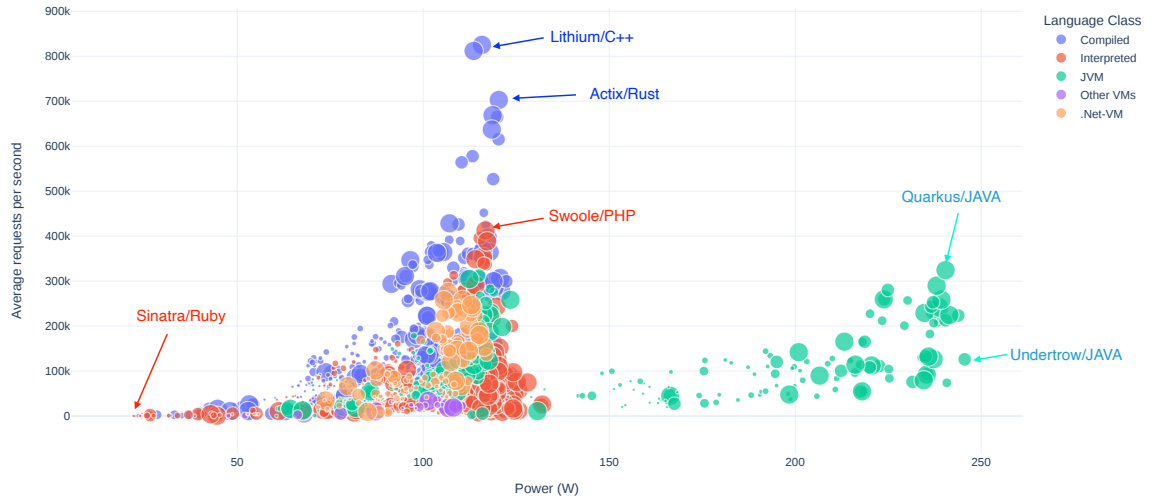


Figure 4.17: Total request Vs average power consumption for the *single query* benchmark (size of circles represents the number of clients)

impact on the average power consumption. Moreover, the overall average power consumption decreased by 20 Watts.

Figure 4.21 reports on the number of RPS per level for each framework. Swoole dropped in the term of performance, while reducing the average power unlike compiled languages based framework, such as lithium and actise.net gained in term of performances while keeping the same power consumption. Another interesting observation is the linearity between the drop in performance when it comes to requests that contains more than 10 rows. This drop comes with a slight decrease in average power.

Plain Text and Json Serialization In this scenario, the client hits its limit before servers, as highlighted in Figures **??**. The ceiling is almost linear for the compiled frameworks and the JVM-based ones. This is also explained by the fact that the high level of stress is on the top, unlike other scenarios.

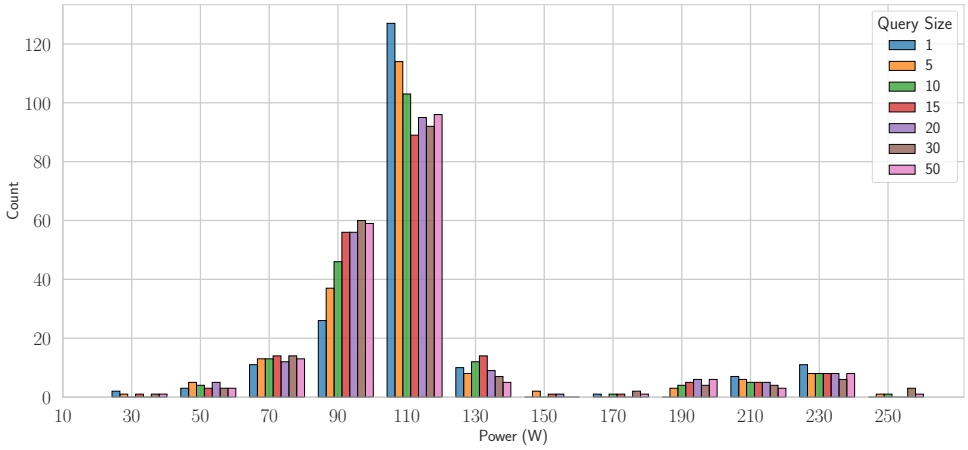


Figure 4.18: Average power consumption for the multiple queries

4.3.5 Threats to Validity*TODO : missing*

We are aware of the bias inducted by the implementation of a candidates, therefore we propose the framework (see the part of extension) to allow the readers to confirm themselves any new hypothesis. Regarding a new framework , an new workload a new database ..etc

4.3.6 Conclusion*TODO : missing*

4.4 Conclusion*TODO : missing*

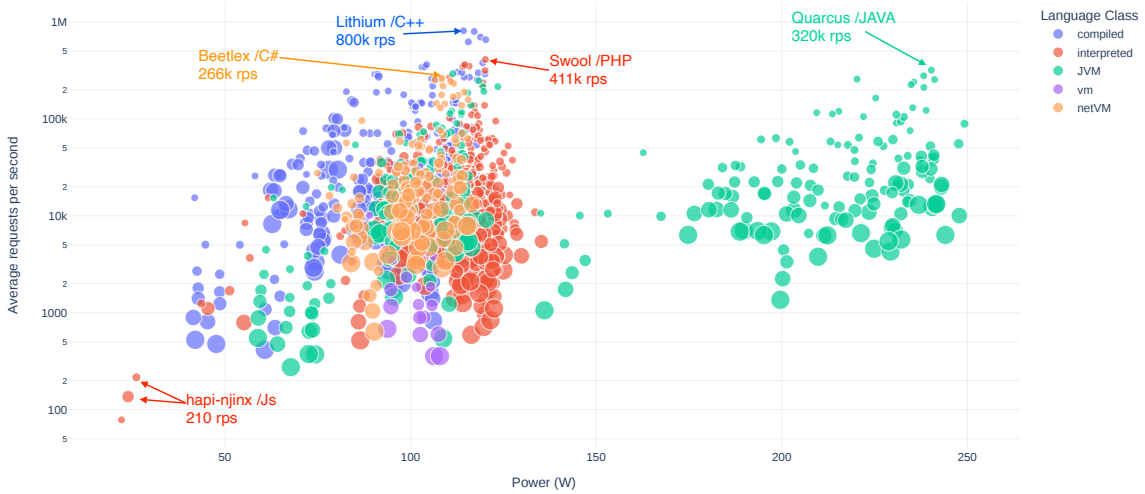


Figure 4.19: Total request vs average power consumption for the single query benchmark (size of circles represents the number of clients)

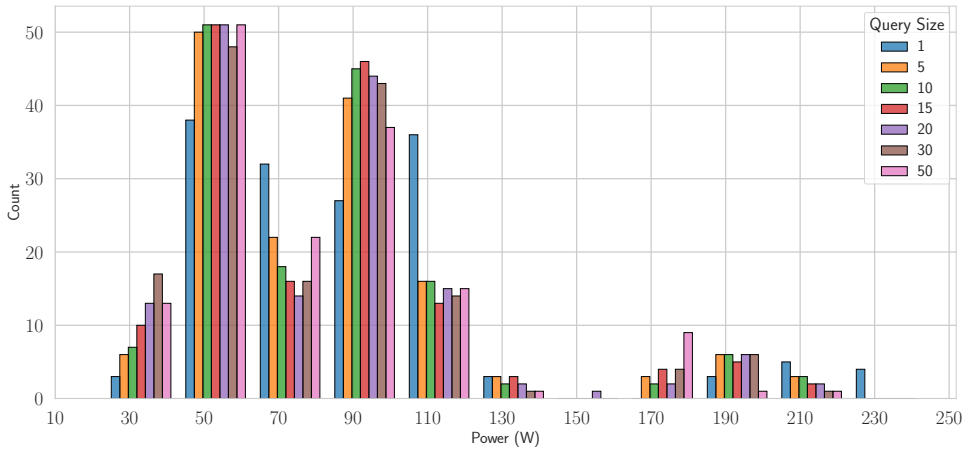


Figure 4.20: Average power consumption for the update benchmark

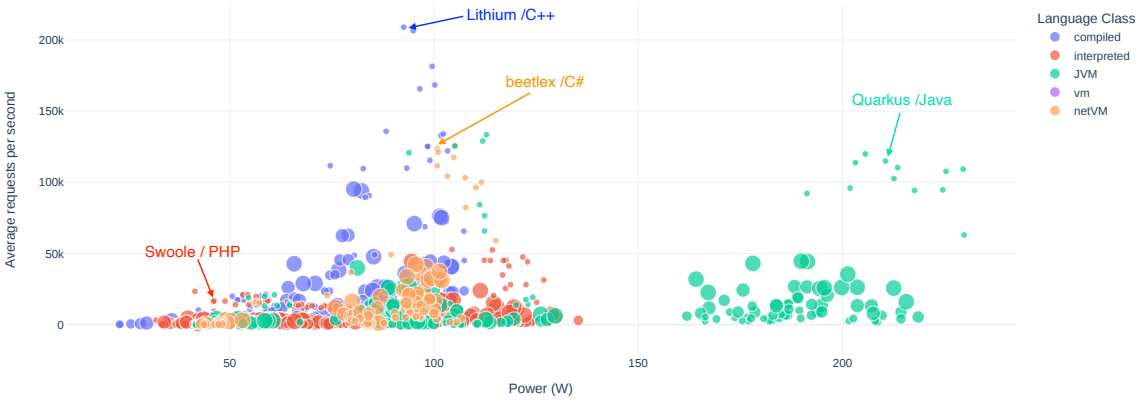


Figure 4.21: Total request Vs average power consumption for the Update benchmark (size of circles represents the number of clients)

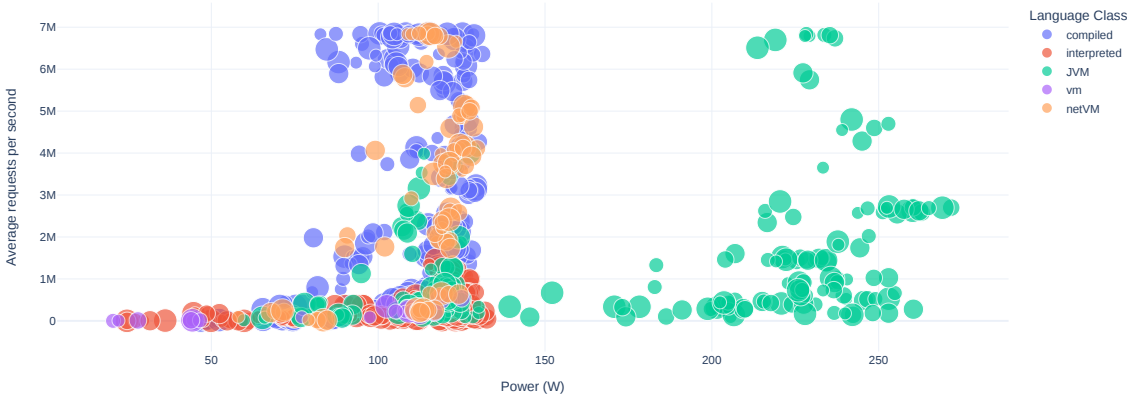


Figure 4.22: Total request Vs average power consumption for plainText benchmark (size of circles represents the number of clients)

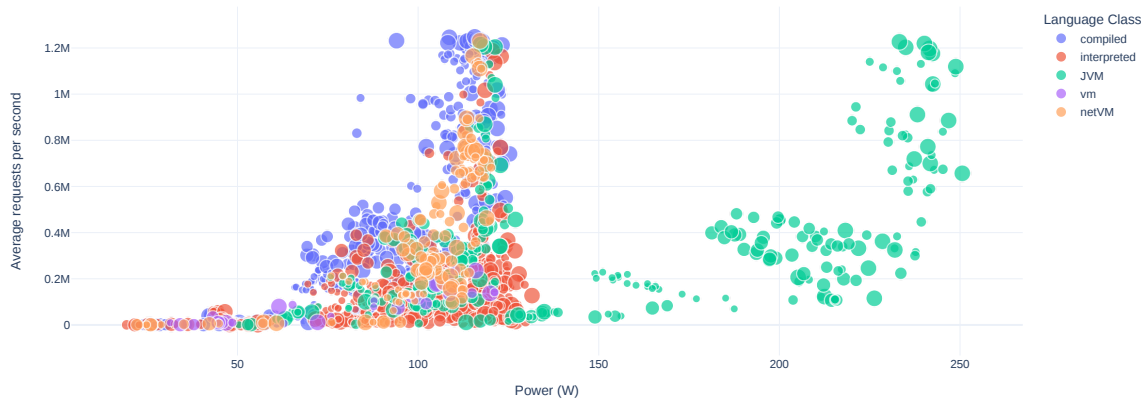


Figure 4.23: total request vs average power consumption for JSON Serialization test (size of circles represents the number of clients)

Part II

Optimizing Application Runtimes

Chapter 5

The Impact of Python Runtime on Energy Consumption

5.1 Introduction

Dynamic programming languages, except Perl, have surpassed compiled programming languages in terms of popularity among software system developers over the past decade (cf. Figure 5.1). However, it remains unclear whether this category of dynamic programming languages can truly compete with compiled ones in terms of power consumption.

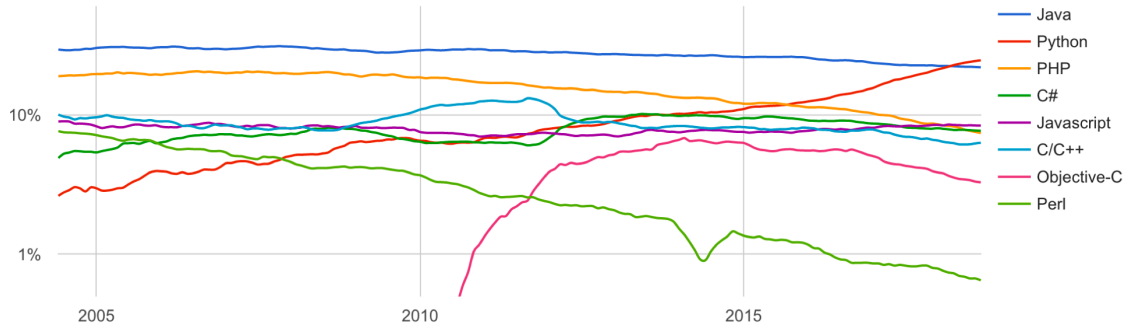


Figure 5.1: PYPL Popularity of Programming Languages [2].

In particular, Noureddine et al. in 2012 [81], and then Pereira *et al.* [92] in 2017, conducted empirical power measurements on this topic, and both concluded that compiled programming languages overcome dynamic ones when it comes to power consumption. According to their experiments, an interpreted programming language, like Python, can impose up to a 7,588 % energy overhead compared to C [92] (cf. Figure 5.2).

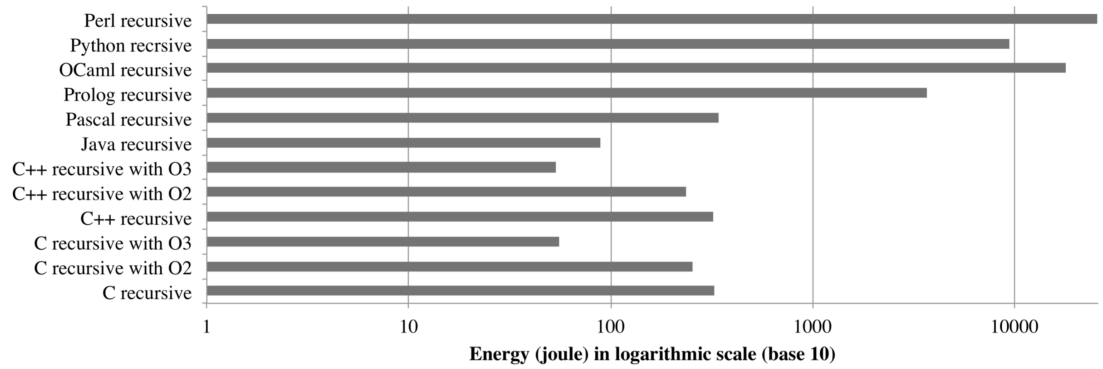


Figure 5.2: Energy consumption of a recursive implementation of Tower of Hanoi program in different languages [81]

In this chapter, we explore the oblivious optimizations that can be applied to Python legacy applications to reduce their energy footprint. As Python is widely adopted by software services deployed in public and private cloud infrastructures, we believe that our contributions will benefit a wide diversity of legacy systems and not only favorably contribute to reducing the carbon emissions of ICT, but also reduce their cloud invoice for the resources consumed by these services. More specifically, this chapter focuses on runtime optimizations that can be adopted by developers to leverage the power consumption of Python applications. We start by studying the impact of programmer choices, such as the type of data or control structures, on the global energy consumption of the execution code. Then, we discuss other factors, such as the levels of concurrency, before we investigate other non-intrusive approaches to optimize the energy consumption of applications. One type of optimization includes alternative interpreters and libraries that are dedicated to optimizing the code without changing its structures, such as *ahead-of-time* (AOT) compilation and *just-in-time* (JIT) libraries that are maintained by the community.

5.2 Motivation

5.2.1 Python Popularity

Nowadays, Python attracts a large community of developers who are interested in data analysis, web development, system administration, and machine learning. According to a survey conducted in 2018 by JetBrains,[1] one can fear that the wide adoption of dynamic programming languages in production, like Python, may critically hamper the power con-

[1]https://www.jetbrains.com/research/python-developers-survey-2018/

sumption of ICT. As the popularity of such dynamic programming languages partly builds on the wealth and the diversity of their ecosystem (*e.g.*, the NumPY, SciKit Learn, and Panda libraries in Python), one cannot reasonably expect that developers will likely move to an alternative programming language mostly for energy considerations. Rather, we believe that a better option consists of leveraging the strength of this rich ecosystem to promote energy-efficient solutions to improve the power consumption of legacy software systems.

5.2.2 Python Gluttony

According to [94] and [81], Python tends to be one more energy hungry programming language. As one can notice in Figure 5.2, Python consumes 30 times more than C or C++. The benchmark was done with an implementation of the Tower of Hanoi[2] of 30 disks.

As shown in Table 5.1, one can observe that, for most of the applications taken for the *Computer Language Benchmark Game* (CLBG), Python takes more time to execute—the only case that he was not the worst one was in the benchmark regx-redux where he beats Go—and in some cases the gap was huge, such as in n-body where Python took around 100 times more than C++.[3]

Python consumes energy, mainly because it is slow in execution. Its flexibility and simplicity caused it to drop off in performance because Python gains its flexibility from being a dynamic language. Therefore, it requires a faster interpreter to execute its programs to compete against alternatives written in compiled programming languages, such as C and C++ or semi-compiled languages like Java.

Table 5.1: Comparison of CLBG execution times (in seconds) depending on programming languages.

	C	C++	Java	Python	Go
pidigits	**1.75**	1.89	3.13	**3.51**	2.04
reverse-complement	**1.75**	2.95	3.31	**16.76**	4.00
regx-redux	**1.45**	1.66	10.5	15.56	**28.69**
k-nucleotide	5.07	**3.66**	8.66	**79.79**	15.36
binary-trees	**2.55**	2.63	8.28	**92.72**	28.90
fasta	**1.32**	1.33	2.32	**62.88**	2.07
Fannkuch-redux	**8.72**	10.62	17.9	**547.23**	17.82
n-body	9.17	**8.24**	22.0	**882.00**	21.00
spectral-norm	1.99	**1.98**	4.27	**193.86**	3.95
Mandelbort	1.64	**1.51**	6.96	**279.68**	5.47

[2]https://en.wikipedia.org/wiki/Tower_of_Hanoi
[3]https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

5.2.3 Python Limits

To reduce the energy consumption of Python, we started by targeting the main usage of this programming language, which is revealed to be data science and web development. Figure 5.3 illustrates a study published by the JetBrain company on Python developers.[4] 57% of the respondents reported that they use Python for data science, 51% said they are using it for web development, and around 40% are using it for system administration.[5]

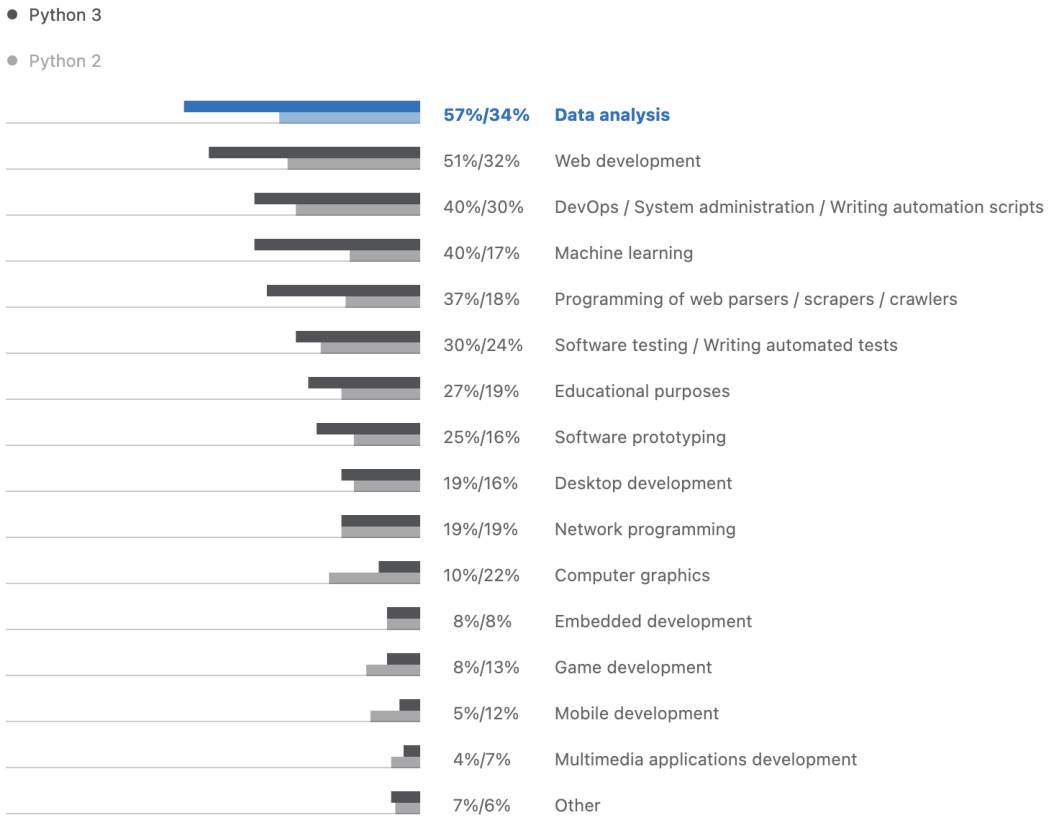


Figure 5.3: Use cases of Python (source: JetBrain).

In this chapter, we investigate the energy footprint of Python in its most popular domains of adoption. We first explore the data and control structures, aiming to reveal some fundamental guidelines, as Hasan et al. did in [48]. Then, we measure the energy consumption of several Python implementations to propose a non-intrusive technique to improve energy efficiency. Therefore, this chapter will address the following research questions:

[4]https://www.jetbrains.com/lp/python-developers-survey-2020

[5]The options in this survey were not mutually exclusive.As a result, the total of the percentages is greater than 100%.

RQ 1: *What is the energy footprint of Python when used in data science?*

RQ 2: *Are the Python guidelines energy-efficient by construction?*

RQ 3: *Can we reduce the energy consumption of Python programs without altering the source code?*

To answer these questions, we report on 4 case studies that intend to answer these research questions. First, we study the energy behavior of Python in two application contexts: web applications (cf. Section 5.3) and machine learning. Then, we dive deeper into the energy consumption of Python core structures, before concluding with the impact of the Python interpreter on energy consumption.

5.3 Green Web Development

Django[6] and Flask[7] are the most popular frameworks for web development. According to a Jetbrains poll,[8] Django is used in 40% of the cases, while Flask is used in 41% of the cases.

In contrast to Flask, which is a micro-framework, Django is a high-level web framework that provides a standard method for creating and maintaining complex and scalable database-driven websites quickly and effectively. Therefore, this study will focus on the latter one.

5.3.1 Life-cycle of a Request in Django

Django is a MVT (*Model-View-Template*) framework, which means that it follows the MVC (*Model-View-Controller*) pattern. Figure 5.4 describes the life-cycle of a request in this framework. Whenever a request arrives in Django, it is processed by the middleware layers, one at a time. These middleware layers are in charge of authentication, security, and so on. Once the request is processed by these layers, it is passed to the URL router, which extracts the URL from the request and tries to match it to the defined URLs. After getting the matching URL, the corresponding view function is called. This function is responsible for treating the request, gathering the data, and then generating the response that will be put inside a template to be returned as a response. As Django adopts a MVT model, it also offers an automatic way to retrieve data from the database to the view using the *Object Relational Mapping* (ORM) [87].

First, we start by investigating the energy consumption of the request-response life cycle in Django, to determine which layer consumes the most energy. To do so, we created a sample Django application that returns the response of the request. We tracked its energy

[6]https://www.djangoproject.com/
[7]https://flask.palletsprojects.com/
[8]https://lp.jetbrains.com/python-developers-survey-2021/

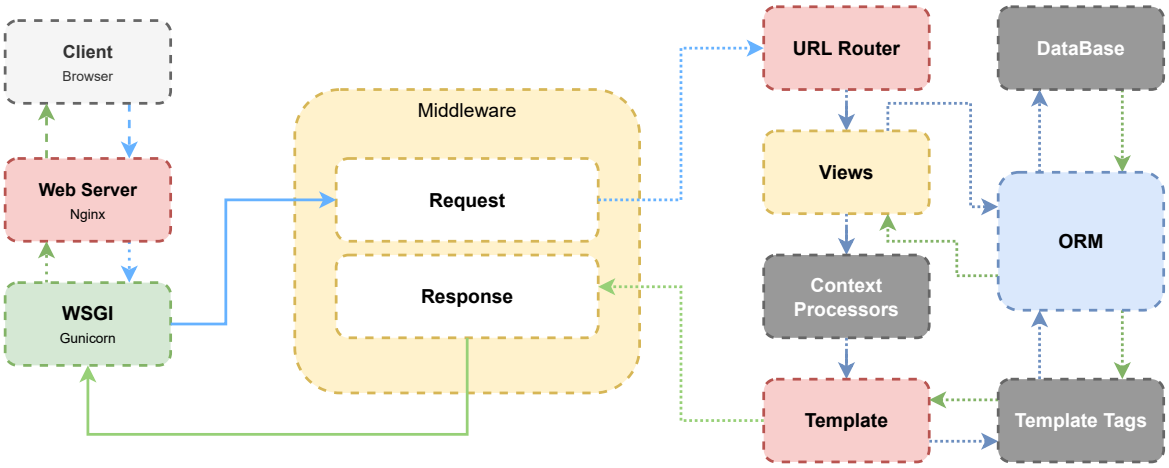


Figure 5.4: Request-Response life cycle in Django

consumption using JouleHunter.[9] JouleHunter is an open-source library that I developed to help practitioners to identify energy hotspots in their applications using statistical profilers. In the case of Django, it can be added as an extension with no additional setup or change to the source code. The energy consumption of the request-response life-cycle in Django is shown in Figure 5.5. As we can notice, 91.4% of the total energy consumption is spent to resolve the request by retrieving the data, while only 5.27% is spent to render the response.

Figure 5.5: Tree representation of the energy consumption of a single request in django (naive version)

Therefore, we chose to study if the choice of the database and the ORM impacts the energy consumption and the performance of the website. To accomplish so, we examined the cost of a single request of the prior website using various ways to extract data from the database. We considered using two different databases, POSTGRESQL and SQLITE3, that store the same records, and three different ways to fetch the records:

1. Vanilla relies only on the ORM to retrieve the data,

[9] https://github.com/powerapi-ng/joulehunter

2. Prefetch queries the data before being requested,

3. Optimized leverages SQL without passing by the ORM.

As one can observe in Figure 5.6, the strategy to query stored data has a huge impact on energy consumption. As the Vanilla strategy can consume up to 10× more energy than the Optimized one. Conversely, the choice of the database does not exhibit a key impact on the total energy, despite their different behavior regarding the execution time and the average power. This can be useful to support developers in choosing which database engine they can adopt, guided by the number of expected requests and the targeted performance.

Another interesting observation is the impact of the interpreter, as Figure 5.6 highlights. For example, using the PyPy interpreter reduces the energy consumption, even when adopting the Vanilla strategy.

Finally, we run the same experiment with the Optimized strategy using JouleHunter. Figure 5.7 depicts the resulting energy consumption. As one can notice, while the rendering method consumed the same amount of energy as in the Vanilla strategy (around 1.3 kJ), the resolve part dropped by 20×.

We can therefore conclude that the database and ORM selection have a major impact on the website's performance and energy consumption. And, unlike the rendering portion, there are many options available for the database, the ORM, and the data handling strategy.

5.4 Python Insights

This section aims to formulate some actionable insights to optimize the energy consumption of Python applications without altering the source code. Therefore, we start by extending the work of Hasan et al. and **(author?)** to the context of Python environments [48, Oliveira et al.].

5.4.1 experiment and results

Another field of investigation is the type of data and control structures that might impact the energy consumption. We iterate over a list using 3 methods. First, the classical `for(i in range(len(n)))`. However, as we can see here, unlike other programming languages, it requires extra operations, such as determining the length of the collection and then using the iterator range. So, we tried the more adapted version `for(element in collection)`. Moreover, in most programming languages. the `for` loop is translated to a `while` loop (transformation from D type to B type – asm –), therefore we wanted to compare this with a `while` version. After determining the main ways to iterate over a `loop`, we run the collection

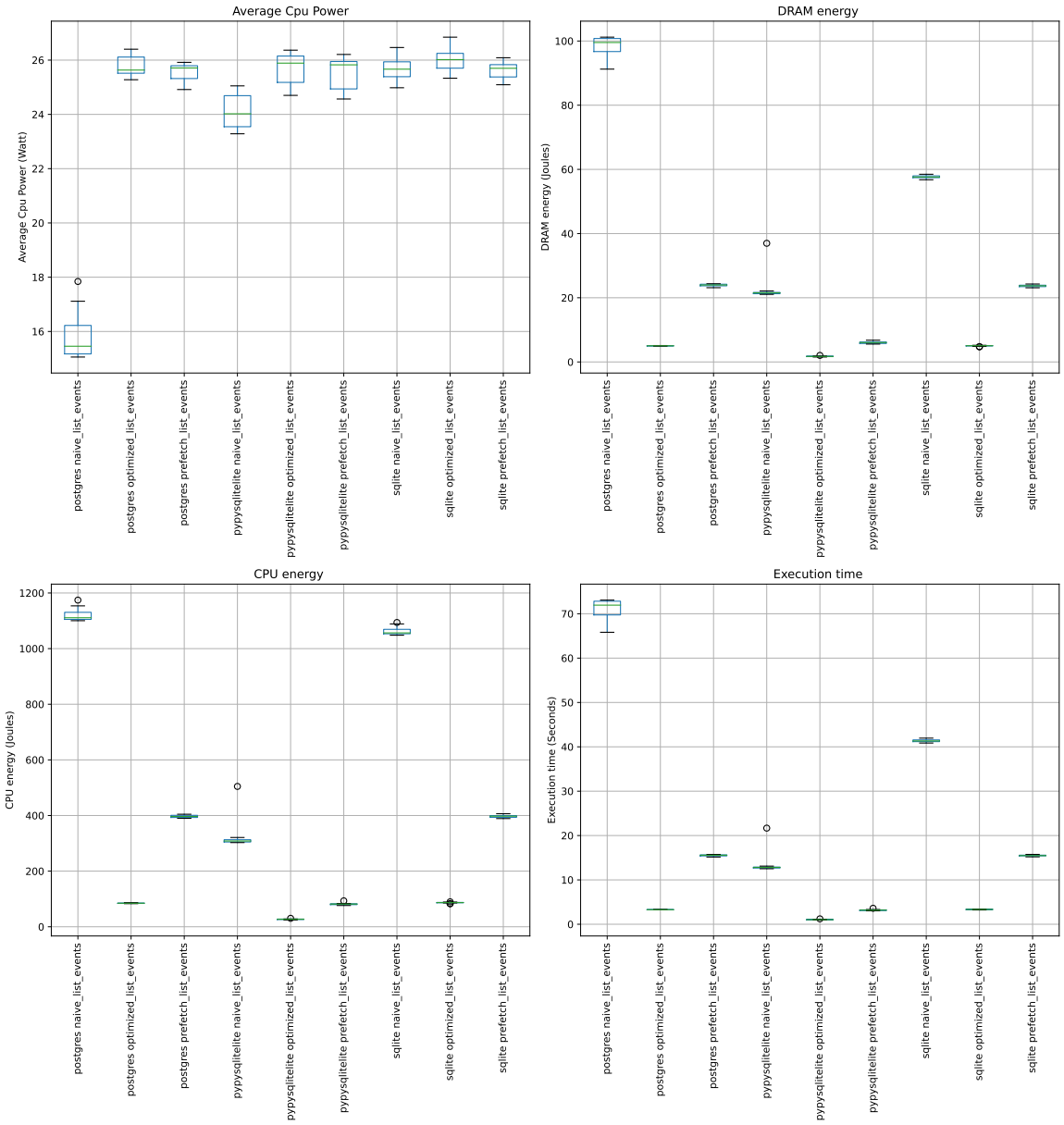


Figure 5.6: Energy behavior resulting from data access strategies.

algorithms of different data types to observe if they impact the energy consumption of the code, and we repeated it for the size of the collection.

As one can see in Figure 5.8, the type of the data has no impact on the energy consumption. However, the way one iterates over the collection has a huge factor. Interestingly, the `for in range` loop was by far the optimal one, followed by the regular `for in collection`, and the `while` part was the last one with an overhead of 400% compared to the first option.

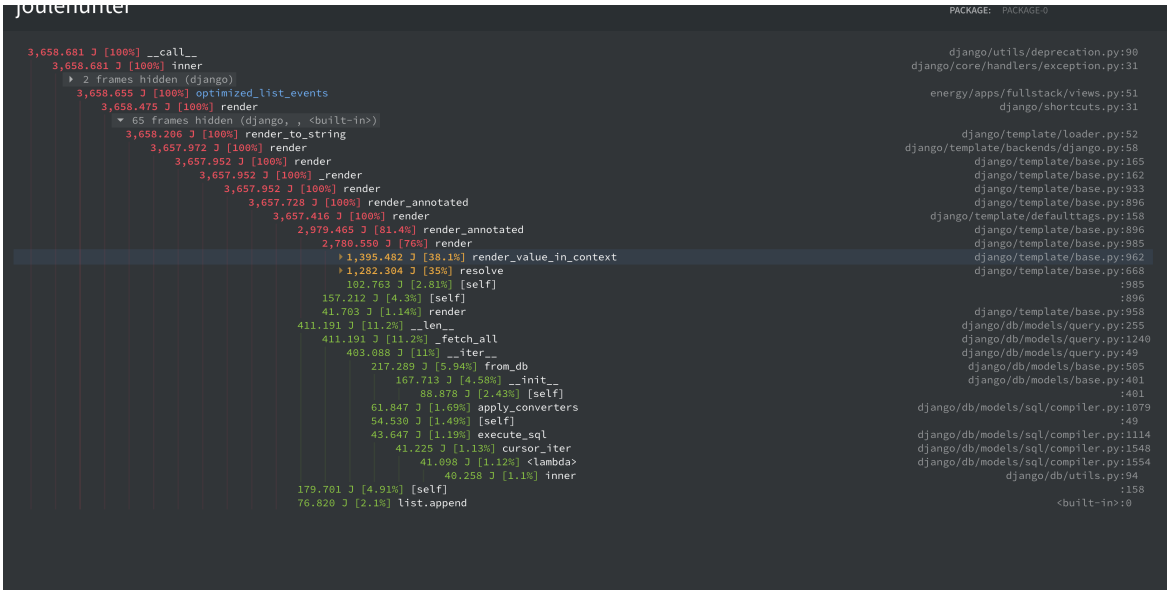


Figure 5.7: Tree representation of the energy consumption of a single request in Django (naive version)

The reason behind such a behavior is mainly related to how the Python interpreter is implemented[10],[11]. To reduce the latency of Python applications, most of the built-in functions and operations are written in C, and the same goes for the function `range`.[12] Furthermore, the function `len` has a complexity of $\mathcal{O}(1)$ as it is based on the function `Py_SIZE` of C, which stores the length in a field for the object [13]. Therefore, the `for in range` is creating a new iterator that has the same length as the first one and, for each iteration, requires second access (`l[i]`) instead of one—explaining the doubled time. The `while` is even slower due to the implicit increment of the variable, which causes an extra operation during the loop. To confirm this hypothesis, we tried to construct a new list by editing the elements of the previous one (cf. Figure 5.9). And, as predicted, the built-in methods are the most energy-saving ones, while the customize `while` loop is the heaviest.

Another interesting finding is the impact of anonymous functions (also known as lambda expressions) on energy consumption. The reason is that Python treats these functions as local variables, unlike the predefined ones which are global in our case. Therefore, they are faster and consume less energy.

[10]https://www.python.org/doc/essays/list2str
[11]https://www.pythonpool.com/for-vs-while-loop-python/
[12]http://python-history.blogspot.com/2010/06/from-list-comprehensions-to-generator.html?m=1
[13]https://wiki.python.org/moin/TimeComplexity

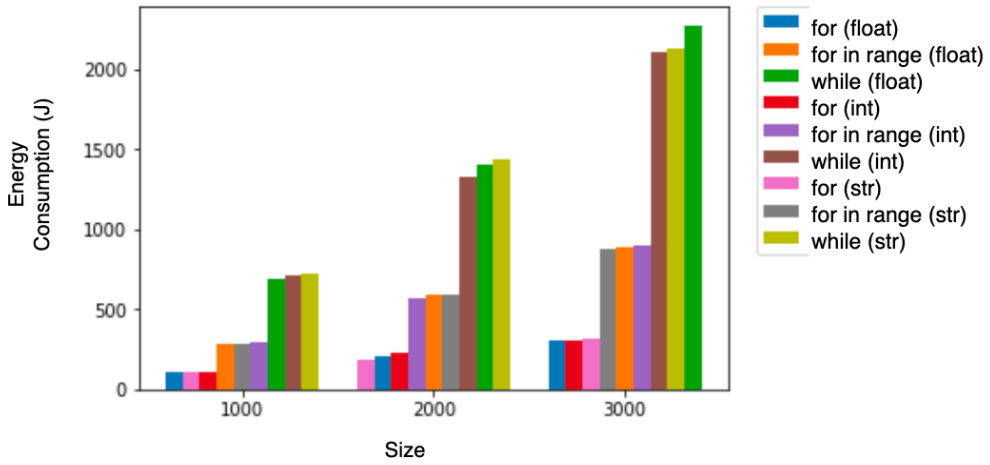


Figure 5.8: Comparison of the energy consumption of different Python loops.

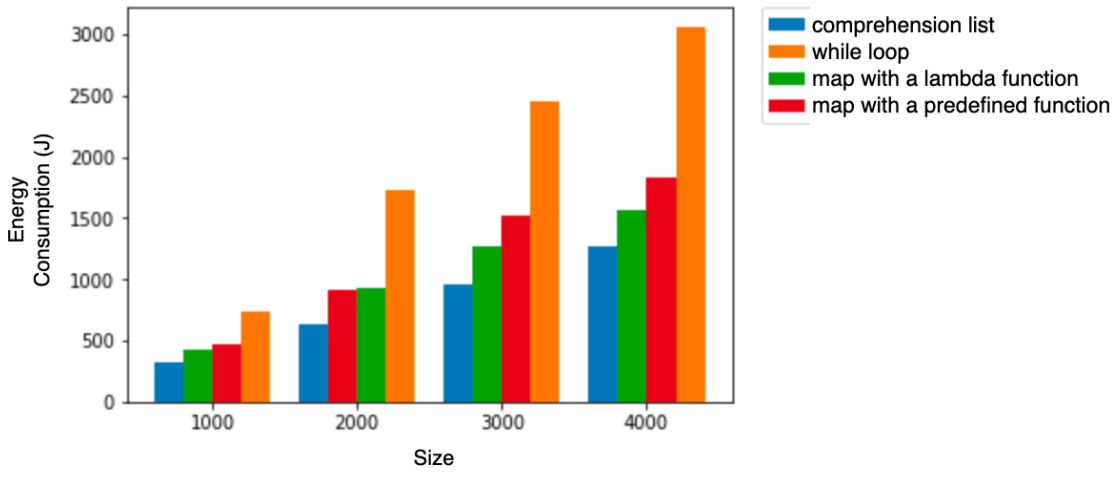


Figure 5.9: Comparison of the energy consumption of different methods to convert a list.

5.4.2 Synthesis

This study demonstrates that the optimal way to reduce the energy consumption of Python application is to follow the guidelines and to privilege the built-in functions.

5.5 Python & Multiprocessing

Cost of Parallelism in Python

Additionally, Figure 5.11 reports on the energy consumption of Python applications when introducing parallelism. Python applications are single-threaded systems constrained by the *global lock system* (GLS). However, due to the increase of cores/threads per CPU, many Python libraries started to take advantage of this hardware feature by allowing concurrent execution of Python processes. For example, the multiprocessing library[14] spawns sub-processes to increase the degree of concurrency of Python applications. As one can see in Figure 5.11, the energy consumption is correlated with the number of threads until reaching the limit of physical cores, when concurrent processes start to compete for the CPU.

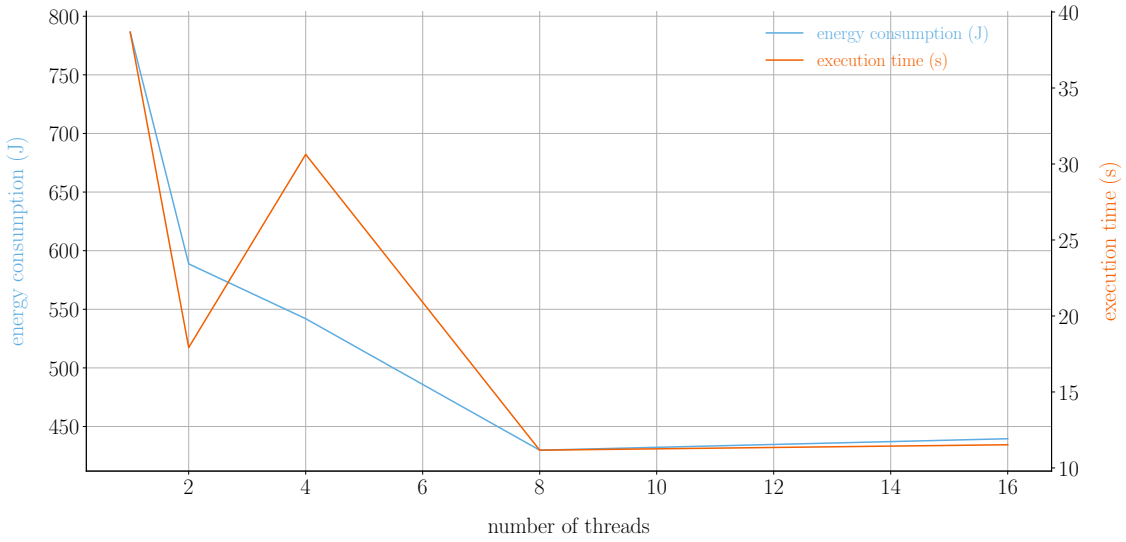


Figure 5.10: Energy consumption of Python multiprocessing depending on the number of exploited threads.

Before reaching the limit of physical cores, we also observe that the scheduler of the operating system tends to favor the execution of processes on the same physical core, by taking advantage of the hyper-thread feature. While this strategy aims to save energy

[14]https://docs.python.org/3/library/multiprocessing.html

by leveraging the ACPI P-states and C-states of unallocated cores, this leads to increase execution times.

Figure 5.11 compares the behavior of python programs when we try to introduce parallelism. As we know python is a single-threaded program thanks to the GLS (*Global Lock System*) however, due to the increase in the number of cores/threads per CPU, many libraries started to take advantage of this hardware feature by simulating multithreading as multiple instances of the processor.

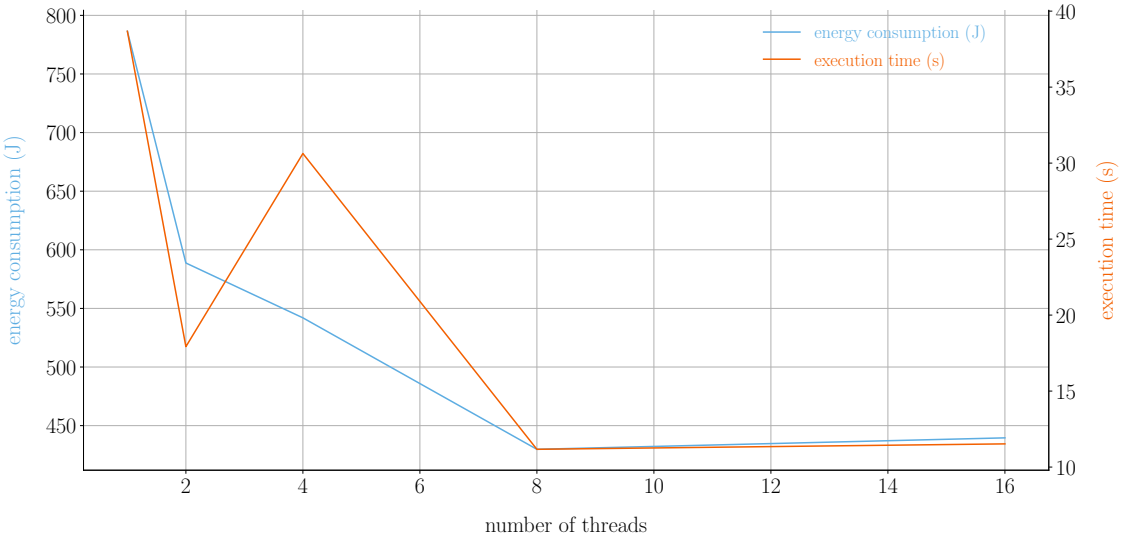


Figure 5.11: Energy behavior when using multiprocessing.

As we can see in Figure 5.11, the energy consumption correlates with the number of threads until reaching the number of physical cores, then losing the benefit from multiprocessing by facing concurency issues imposed by different sub-processes competing for the CPU resources. Another finding is observed when hitting the number of physical cores: there is an increase of execution time, but with a reduced energy. The reason behind this is the scheduler of the operating system. More specifically, it favors hyper-threads instead of physical cores, hence increasing context switches which cause the performance bottleneck. Howerver,the other two physical cores are not consuming energy, neither their hyper-threads, which explains the energy savings. In the Chapter , we discuss this behavior of the scheduler more deeply and we assess that it is not due to Python, but is a more generic behavior.

5.6 Python & Machine Learning

Machine learning is becoming an integral part of our daily lives, growing more potent and energy-hungry each year. As machine learning can have a significant impact on climate change, it is vital to investigate mitigation techniques.

5.6.1 Experimental Protocol

Measurement context

Hardware settings Chifflot 8 from Grid 5000's Lille site was used for all of the trials. The machine is outfitted with two Intel Xeon Gold 6126 CPUs, each having 12 physical cores, 192 GB of RAM, and two 32 GB Tesla V100 GPUs.

Software settings For the sake of reproducibility, each experiment is done within a Docker container using Jupyter lab. These tests are run on top of a minimal version of Debian-10 to increase the accuracy of the tests by eliminating any unnecessary processes.

Input Workload

Models Several models were developed, however, only two were used in the final trials because they achieved 94 percent accuracy in an acceptable length of time. David Page's cifar10-fast [15] and Woonhyuk Baek's torch skeleton [16] are shown here.

Datasets The CIFAR-10 dataset was the major source of data for the studies. It is made up of 60000 32x32 color images grouped into ten categories.

Some experiments were done using the MNIST dataset of handwritten digits to validate the results acquired from the first dataset. The model did not need to be updated because the 60000 28x28 grayscale photos were padded.

candidates

The experiments were run with several different CPU and GPU configurations:

- with and without GPU,
- with and without CPU hyper-threading,
- different number of CPU physical cores.

[15] https://github.com/davidcpage/cifar10-fast
[16] https://github.com/wbaek/torchskeleton

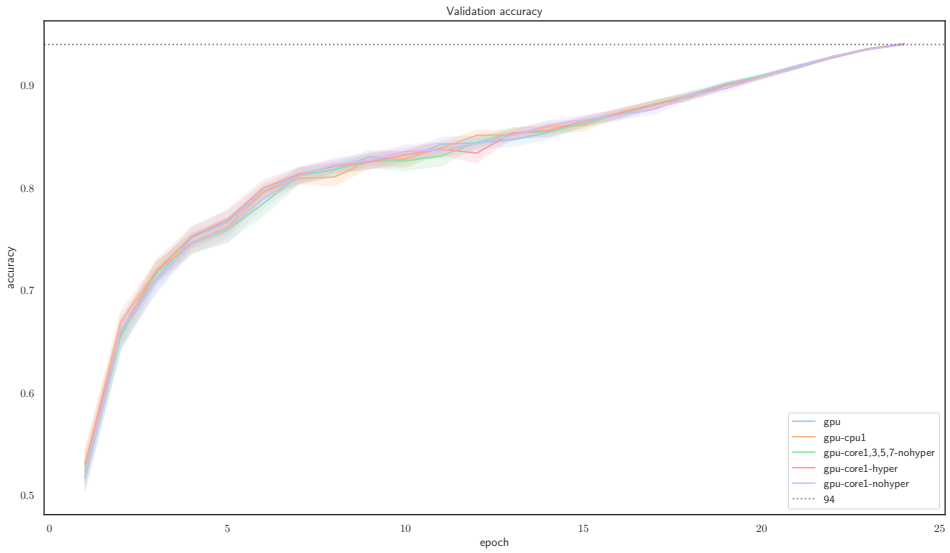


Figure 5.12: Accuracy along epochs.

Key Performance Metrics

We used Pytorch 1.10.0 to train those models and pyjoules to measure the energy consumption of the GPU and CPU.

- accuracy : in %
- execution time : in seconds for both the duration of each epoch and the total duration to a chieve a certain accuracy
- total energy consumption : for the CPU and the GPU

5.6.2 Results & Findings

As the model's accuracy increases, so does the energy required for the next accuracy increment.

Figure 5.13 depicts how the curve steepens as training progresses. For example, training to 90% accuracy requires three times the energy required for training to 80% accuracy.

5.6.3 Conclusion

We discovered that when the model's accuracy improves, so does the energy required for a subsequent accuracy gain. This begs the question of when we should discontinue training.

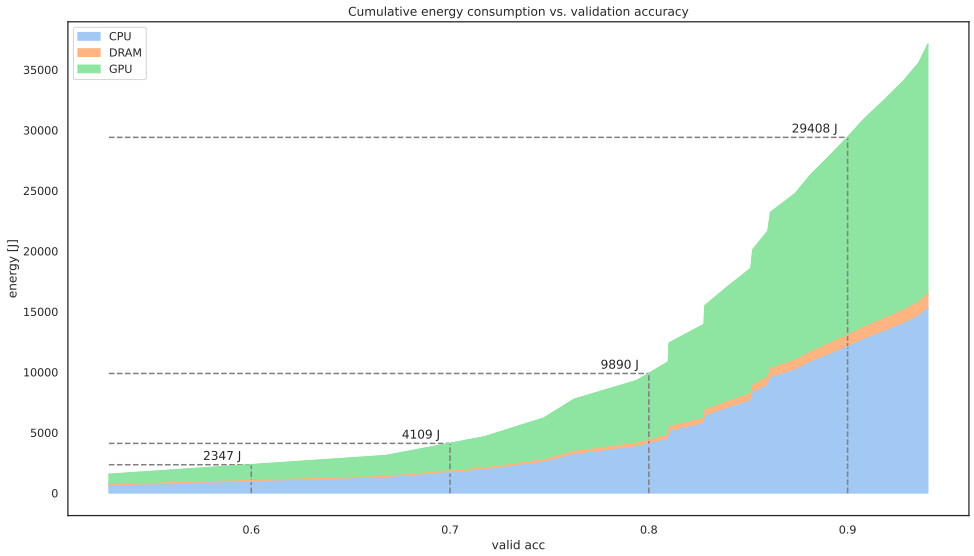


Figure 5.13: Cumulative energy consumption vs accuracy.

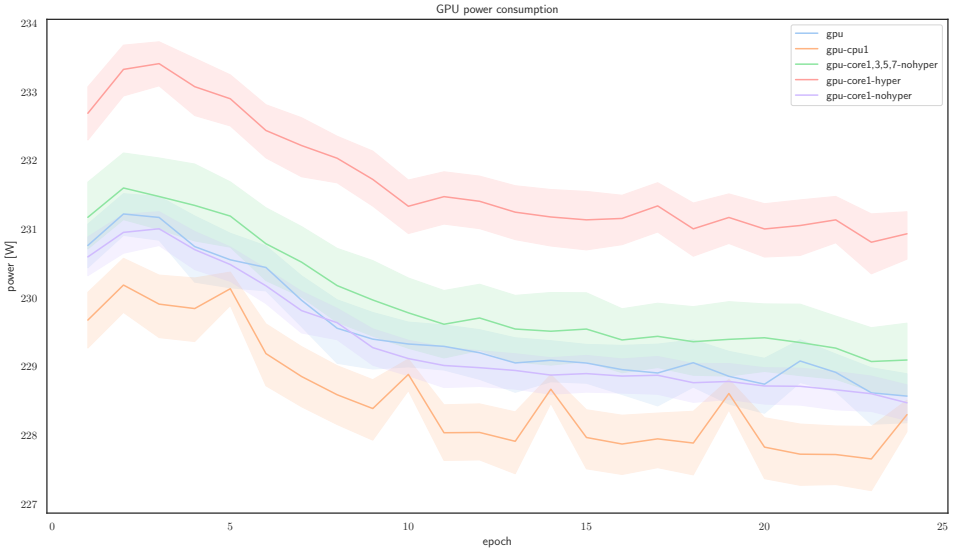


Figure 5.14: Evolution of average GPU power along epochs.

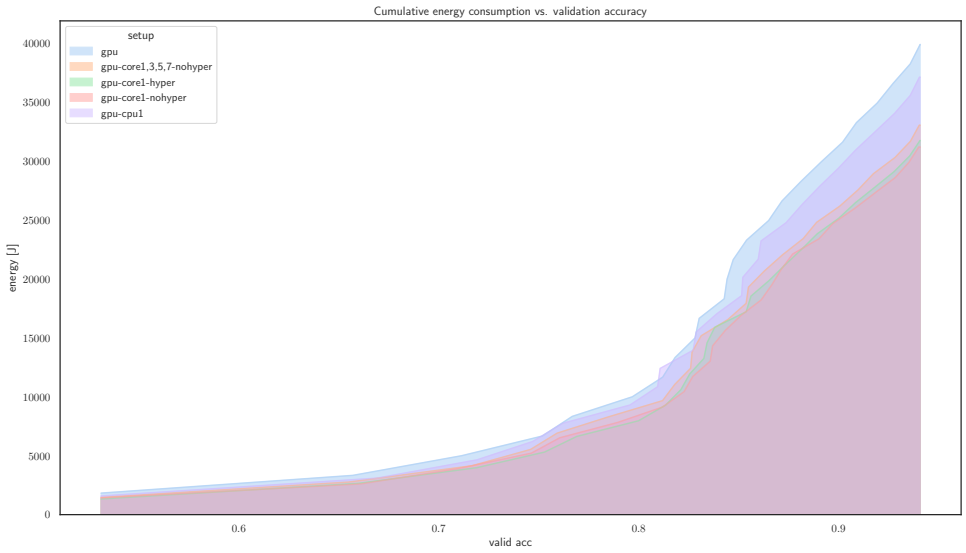


Figure 5.15: cumulative energy of fast10 benchmark within different configurations

Is a 10% increase in accuracy worth it if we have to spend three times the energy? Even while utilizing a GPU consumes more total power, the training time is significantly reduced. As a result, the utilization of a GPU is required to lower the energy consumption of the training. Some experiments revealed unusual behavior that was not studied during this internship. These could be examined in future studies and perhaps used to achieve more energy savings. The execution time did not depend on the number of cores employed in several studies. This parameter had a significant impact on training time in others. As a result, the best core configuration will be determined by the script. The next steps in this area could include exploring the reasons for these discrepancies and possibly creating a script that can automatically find the ideal core configuration for a specific script.

5.7 Python Interpreters

Due to the lack of support for most non-conventional Python interpreters, we mainly focus on micro-benchmarks. Except for PyPy, most of the Python implementations do not support extra Python libraries, despite those extra implementations being developed to optimize a specific library, such as Numba with Numpy, or intelpython with machine learning algorithms.

5.7.1 Preliminary studies

For the first studies, we used the official version of Python, because the goal was mainly to highlight the impact of the structure of code on energy consumption. One main drawback of the previous method is the work to be done to update the existing code base to reduce energy consumption. To avoid such hustle, we tried to find a non-intrusive approach to make the Python code more eco-friendly without altering its structure. Python is an interpreted language, which led many initiatives to implement their own interpreter to improve one or many aspects of the Python code. In the following section, we discuss the impact of those implementations on the energy consumption of python programs, and in which case, one should use a non-conventional interpreter to save the energy consumption of their application.

To do so, we gathered a list of interpreters, transpilers and other optimization libraries that can contribute to reduce the energy consumption of legacy Python applications:

1. **CPython**:[17] This Python interpreter, written in C, is the reference interpreter of Python. CPython compiles the source code into byte-code and then interprets it. The CPython project supports both versions of Python 2 and 3;

2. **PyPy**:[18] An alternative implementation of the Python interpreter. It is written using *RPython* to use the JIT. It compiles the most used portions of the Python code into a binary code for better performance. To benefit from these optimizations, the program has to be executed for at least for few seconds so the JIT has enough time to warm up, the JIT optimization are only applied to the code written by the developer and not to external libraries;

3. **Cython**:[19] A static compiler for Python. It translates the Python code into C, and then compiles it using a C compiler. It also supports an extended version of the Python language that allows programmers to call *C functions*, declare *C types* and use static types, which will help the translation of Python objects into native types, such as integers, float. This often means better performances, since native C libraries are almost all the time faster than the Python written once [92];

4. **Intel Python**:[20] A customized interpreter developed by Intel to enhance performances of Python programs. It is dedicated to data sciences and high-performance computing.

[17] https://www.python.org/
[18] http://pypy.org
[19] https://github.com/cython/cython
[20] https://software.intel.com/en-us/distribution-for-python

It uses some Intel kernel libraries, such as Math Kernel Library (Intel MKL[21]) and data analytics acceleration library (Intel DAAL[22]). It supports both versions of Python;

5. **Active Python**:[23] It is developed by the Activestates company and provides a standardized Python distribution to ensure license compliance, security, compatibility and performance. Therefore, ActivePython implements its built-in packages (more than 300 packages) and supports both versions of Python;

6. **IronPython**:[24] A .Net-based Python interpretation platform written in C# that is used with the .Net virtual machine or Mono. It benefits from all the optimizations of .Net virtual machines, such as the JIT and garbage collector mechanisms;

7. **GraalPython**:[25] A Python interpreter that is based on GraalVM[26] (a universal virtual machine developed by oracle for running applications written in different programming languages). For the time being, it only supports Python 3 and it is still in the experimental stage;

8. **Jython**:[27] An implementation of Python programming language written in Java for the *Java Virtual Machine* (JVM). Similar to IronPython and GraalPython, it leverages the optimization mechanisms provided by the JVM to enhance the Python performances;

9. **MicroPython**:[28] A lightweight Python version dedicated to embedded systems and micro-controllers;

10. **Nuitka**:[29] A Python compiler written in Python that generates a binary executable from Python code. It translates the Python code into a C program that is then compiled into a binary executable;

11. **Numba**:[30] A library that includes JIT compiler to enhance the performances of Python functions using the industry-standard LLVM compiler library;

[21]https://software.intel.com/en-us/mkl
[22]https://software.intel.com/en-us/intel-daal
[23]https://www.activestate.com/products/activepython/
[24]https://ironpython.net
[25]https://github.com/graalvm/graalpython/
[26]https://www.graalvm.org/docs/why-graal/
[27]https://jython.github.io
[28]http://micropython.org
[29]http://nuitka.net/pages/overview.html
[30]https://numba.pydata.org

12. **Shedskin**:[31] A static transpiler that translates implicitly statically typed python into C++ code;

13. **Hope** [5]: A Python library that aims to introduce JIT compiler into the Python code;

14. **Parakeet** [**?**]: A runtime accelerator for an array-oriented subset of Python;

15. **Stackless Python**:[32] An interpreter that focuses on enhancing multi-threading programming;

16. **Pyjion**:[33] A JIT API for CPython, same purpose as Parakeet and Hope;

17. **Pyston**:[34] A performance-oriented Python implementation built using LLVM and modern JIT techniques. The project is funded by Dropbox;

18. **Grumpy**:[35] A source-to-source transpiler that translates the Python code into Go before being compiled to a binary executable. It also offers an interpreter, called *grumprun*, which can directly execute the Python code. Unfortunately, we cannot use it because the project is already outdated (last commit is in 2017) and it has a lot of limitations in terms of supporting the Python language, such as some built-in functions and standard libraries;

19. **Psyco**:[36] A JIT compiler for Python;

20. **Unladen Swallow**:[37] An attempt to (use) LLVM as a JIT compiler for CPython.

5.7.2 Runtime Classification

Before further proceeding with the list of candidate runtime for Python applications, we propose a classification according to several criteria:

Type refers to the category of runtime infrastructure that supports the execution of a Python application. In particular, we consider 3 types of environments: *Interpreter*, *Compiler* and *Library*; *Interpreter* refers to the class of environment that does not require any preprocessing of Python source code; *Compiler* introduces a compilation phase before

[31] https://github.com/shedskin/shedskin
[32] https://github.com/stackless-dev/stackless/wiki
[33] https://github.com/microsoft/pyjion
[34] https://blog.pyston.org
[35] https://github.com/google/grumpy
[36] http://psyco.sourceforge.net
[37] https://unladen-swallow.readthedocs.io/en/latest/

Table 5.2: Classification of Python implementations

Name	Type	Runtime	Optimisations		Python	
			JIT	GC	2	3
CPython	Interpreter	C	–	–	✓	✓
Intel Python	Interpreter	C	–	–	✓	✓
ActivePython	Interpreter	C	–	✓	✓	✓
PyPy	Interpreter	Python	✓	✓	✓	✓
IronPython	Interpreter	.Net	✓	✓	✓	✓
GraalPython	Interpreter	GraalVM	✓	✓	–	✓
Jython	Interpreter	Java	✓	✓	✓	–
Stackless Python	Interpreter	Python	–	–	✓	–
MicroPython	Interpreter	c	–	–	–	✓
Pyston	Interpreter	LLVM	✓	–	✓	–
Unladen Swallow	Interpreter	LLVM	✓	–	✓	–
Cython	Compiler	C	–	–	✓	✓
Nuitka	Compiler	C	–	–	✓	✓
Shedskin	Compiler	C++	–	–	✓	✓
Grumpy	Compiler	Go	–	–	✓	✓
Numba	Library	C	✓	–	✓	✓
Hope	Library	Python	✓	–	✓	✓
Psyco	Library	Python	✓	–	✓	✓
Pyjion	Library	.NET Core	✓	–	✓	✓
Parakeet	Library	C	-	–	✓	–

the execution of the application. Finally, *Library* requires some modification of the source code;

Runtime refers to the technology supporting the execution of a Python application. This technology can refer to the programming language used to program the interpreter, the target language for a compiler or a library;

JIT optimization refers to the support of *just-in-time* compilation in the runtime infrastructure supporting the execution of the application;

GC optimization refers to the support of *garbage collection* in the runtime infrastructure supporting the execution of the application;

Python version(s) refers to the list of Python source code versions supported by the runtime environment.

There are other implementations that we did not consider because either the project aborted many years ago or it has very limited support for Python features. After the inventory

Table 5.3: Classification of Python implementations

Version	Interpreter	Transpiler/Compiler	Jit library
Python 2	Cpython2 Pypy2 Pyston Ironpython Jython Micropython Pysec StacklessPython	Cython2 Shesdskin Grumpy	Numba 2 Hope Parakeet Psyco Pyjion
Python 3	Cpython3 Pypy3 GraalPython	Nuitka	Numba3

of those implementations, we filtered them. To keep only the versions that are still maintained and support most Python features. We then classified them into 3 categories depending on their integration with the Python code. In Table 5.3, we describe the implementations that we kept, the version of each implementation and its category.

5.7.3 Experimental Protocol

As discussed in the previous chapter, our idea is to design a benchmarking solution that allows practitioners to reproduce and extends our benchmarks. This benchmarking solution is also the one we use to answer the research questions addressed in this manuscript.

Measurement Context

Hardware settings. All our benchmarks have been executed on a Dell PowerEdge C6420 server, whose hardware features are summarized in Table 5.4. The server uses a minimal version of Debian 9 (4.9.0 kernel version) where we install Docker (version 18.09.5).

CPU	Intel Xeon Gold 6130 (Skylake, 2.10GHz, 2 CPUs/node, 16 cores/CPU)
Memory	192 GiB
Storage	240 GB SSD SATA Samsung MZ7KM240HMHQ0D3
	480 GB SSD SATA Samsung MZ7KM480HMHQ0D3
	4.0 TB HDD SATA Seagate
Network	eth0/enp24s0f0, Ethernet, configured rate: 10 Gbps, model: Intel Ethernet Controller X710 for 10GbE SFP+, driver: i40e
	ib0, Omni-Path, configured rate: 100 Gbps, model: Intel Omni-Path HFI Silicon 100 Series [discrete], driver: hfi1

Table 5.4: Benchmarking server configuration.

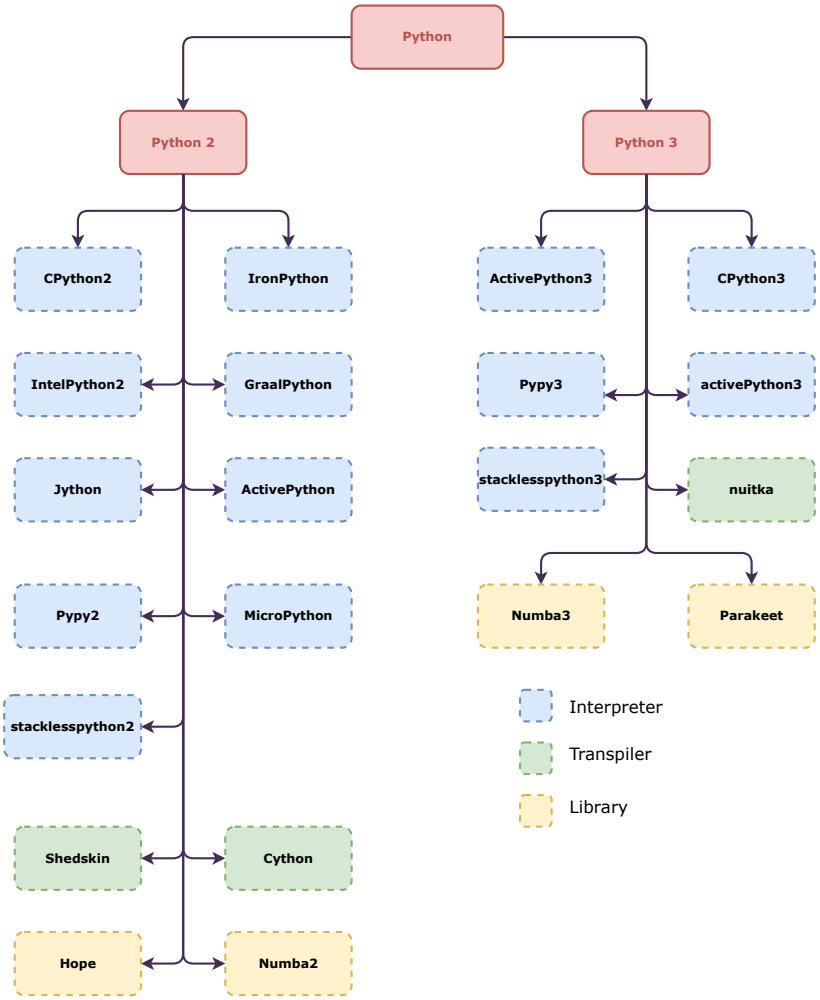


Figure 5.16: Python interpeters

Software settings. For the sake of reproducibility, each experiment runs within a Docker container.

Key Performance Metrics

Our focus will be mainly on CPU energy consumption as it is ten folds more than the DRAM one, since it is a task-based benchmarking, time is highly correlated within the energy, and it will be only useful to explain some specific energy behaviors, thus we do not put a lot of focus on this metric.

Energy measurement. As we know, the energy of a program is integral of its power over time. For our case, we used Intel *Running Power Average Limit* (RAPL) [59] to collect the power samples of the running tests. We run POWERAPI [28], to report on measurements collected by Intel RAPL and upload them to a so-called *computing machine*, then we calculate the Energy using the trapezoidal rule:

$$E = \int_b^a P(t)dt \simeq \sum_{k=1}^{n} \frac{P(t_k - 1) + P(t_k)}{2} \tag{5.1}$$

Figure 5.17 overviews the architecture of our benchmarking infrastructure.

The motivation for separating measurement collection from energy computations is to reduce any interference with the benchmark, our sensor being a lightweight C program running as a Docker container.

Benchmark Preparation

Input workload. To benchmark our implementations, we employed the TOMMTI microbenchmarks suite.[38] TOMMTI is a set of 13 microbenchmarks that examine common language features, such as arithmetic operations, data structures and input/output manipulations, among others. In addition to these microbenchmarks, we implemented some binary operations to investigate the behavior of the previous implementations when it comes to low-level operations that only works with registries.

To study the energy behavior of the Python implementations, we have to focus on the effect of the implementations and mitigate any side effect, such as the organization of the code or any extra consumption due to the operating system or third-party libraries. Therefore, for each benchmark, we took the implementation written in Python2 as a reference and tried to use it in other implementations as it is. If it is not supported by Python3, we converted the

[38] http://www.tommti-systems.de/main-Dateien/reviews/languages/benchmarks.html

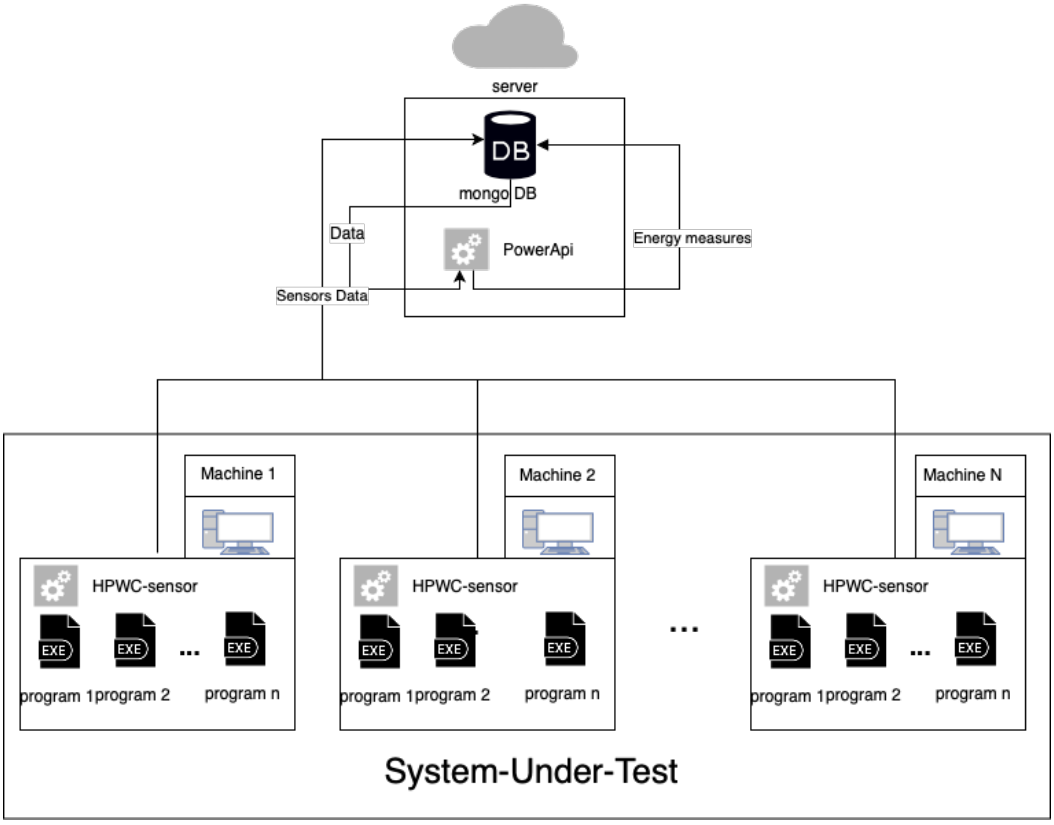


Figure 5.17: Benchmarking architecture deployed with POWERAPI.

code using the official library *2to3*.[39] In the case of the libraries using *JIT* adding a decorator to the function that we want to optimize was enough, if there are other changes, we assume that they alter the original code which is against our purpose. Each benchmark is isolated in a Docker container for several reasons:

- Isolation: each container has only the benchmark program implemented with a single python runtime to remove any interference between different implementations,
- Deployment: to use the benchmarking machine without extra configurations that may alter the behavior of the operating system toward energy consumption,
- Reproducibility: one of the most frequent benchmark crimes [106] in research is the lack of reproducibility—by using Docker we ensure that each benchmark has an image that will be publicly accessible.

Despite the presence of the official docker images for most of the runtimes, we preferred to build our own using the same reference image to remove any bias due to the OS used in

[39]https://docs.python.org/3.7/library/2to3.html

the official images. We used ArchLinux with kernel version 4.9.184 as a base image for all our benchmarks.

Benchmark extension. As we have done with the previous chapters, we provide a tool that allows extending the benchmarks with new workloads and new candidates. In the repository listing the Python implementations under study,[40] we propose a dedicated tool to generate new workloads and new candidates. The script `generator.py` allows practitioners to create new benchmarks by implementing a Python code within different interpreters. Then, it generates `launcher-benchmark.sh` that can be executed to run the associated benchmark. Furthermore, all the successful implementations are stored in a separate directory and the ones that failed (*e.g.*, mostly because of compatibility issues) are stored in a recap file called `benchmarkTest.md`, where `benchmark` is the name of the new workload. To add new candidate runtimes, one should add a base Docker file that contains the new implementation and, if there should be extra manipulation that should be added to the workload files such as adding new decorator or changing some parameters, then they should be added as an extra function in the script `generator.py`. Finally, they should be included in the Python candidates.

5.7.4 Results & Findings

This section will be dedicated to the results of the experiments and statistical analysis of these results. Because we are conducting a comparison study, we first did a Shapiro-Wilk [100] test to see if the data follows a normal distribution. Despite the fact that some implementations had a p-value higher than $a = 0.05$ (such as CPython and ActivePython), other implementations like PyPy and Numba presented a p-values smaller than 0.01, which leads to reject the hypothesis $H0$ that all the distributions follow a normal distribution. Therefore, to compare the different implementations, we performed the non-parametric Mann-Whitney U test [115], with CPython2 and CPython3 as base references.

The Mann-Whitney U test results for each implementation are shown in Table 5.5. The first column displays the implementation's average energy usage, while the second provides the p-value of the test when compared to the reference Python implementation. If the p-value is less than 0.05, the test result is significant. This implies that the given implementation's energy consumption differs significantly from the reference implementation's.As one can notice, most of the implementations differ significantly from the reference implementation.

[40]https://github.com/chakib-belgaid/python-implementations

Table 5.5: Energy consumption of Python runtimes when executing our benchmark.

Implementation	array		intArithmetic		doubleArithmetic		hashes		heapsort		trig	
	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*
ActivePython	402.76	0.008	678.90	0.002	668.16	0.002	977.59	0.008	290.43	0.008	518.32	0.008
CPython2	361.29	base	560.96	base	548.63	base	646.84	base	275.31	base	411.83	base
CPython3	323.90	base	743.64	base	740.50	base	797.60	base	243.32	base	413.60	base
GraalPython	148859.73	0.008	24.63	0.002	24.98	0.002	641.53	0.008	135834.95	0.008	75.21	0.008
IntelPython2	367.50	0.690	579.62	0.015	561.35	0.485	710.23	0.008	268.70	0.690	439.83	0.151
IntelPython3	352.09	0.008	767.40	0.002	765.04	0.026	958.77	0.008	265.97	0.008	479.19	0.008
ipy	305.35	0.008	437.96	0.002	467.42	0.004	1255.44	0.008	256.25	0.016	453.67	0.008
Jython	517.46	0.008	133.04	0.002	160.65	0.002	635.71	0.056	450.76	0.008	630.22	0.008
Micropython	307.76	0.008	821.59	0.002	836.10	0.004	9367.15	0.008	335.25	0.008	532.15	0.008
Nuitka	292.83	0.008	541.95	0.002	543.88	0.004	946.67	0.008	218.31	0.008	390.98	0.008
Numba2	185.59	0.008	27.72	0.002	35.92	0.004	681.30	0.008	2065.55	0.008	444.84	0.008
Numba3	**12.39**	0.008	11.04	0.002	10.76	0.004	920.99	0.008	720.72	0.008	440.99	0.008
PyPy2	17.43	0.008	29.46	0.002	30.34	0.002	**115.53**	0.008	**20.58**	0.008	64.94	0.008
PyPy3	14.53	0.008	17.95	0.002	18.69	0.004	191.64	0.008	**20.47**	0.008	65.27	0.008
Shedskin	47.75	0.008	**7.41**	0.002	**7.37**	0.004	1125.24	0.008	44.97	0.008	**7.92**	0.008

Implementation	longArithmetic		matrixMultiply		io		stringConcat		nestedLoop		except	
	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*
Activepython	661.49	0.008	430.10	0.008	206.48	0.016	15.11	0.310	414.83	0.008	256.64	0.008
CPython2	550.78	base	395.68	base	192.16	base	13.05	base	416.58	base	433.33	base
CPython3	735.87	base	440.09	base	197.98	base	13.95	base	380.24	base	226.73	base
GraalPython	25.66	0.008	45.26	0.008	742.27	0.008	25.18	0.008	11.71	0.008	158.20	0.008
IntelPython2	566.94	0.016	479.78	0.008	200.37	0.421	14.64	0.151	447.08	0.008	469.17	0.008
IntelPython3	778.78	0.008	498.81	0.008	213.12	0.008	14.41	0.310	435.29	0.008	275.28	0.008
ipy	469.11	0.008	416.34	0.008	379.20	0.008	50.13	0.008	336.01	0.008	722.65	0.008
Jython	162.90	0.008	187.38	0.008	198.02	0.310	21.45	0.008	196.06	0.008	733.46	0.008
MicroPython	837.34	0.008	536.38	0.008	5353.12	0.008	43.86	0.008	429.79	0.008	360.71	0.008
Nuitka	542.77	0.008	441.12	0.421	209.21	0.151	12.74	0.310	360.67	0.008	224.42	0.095
Numba2	35.13	0.008	401.02	0.310	218.04	0.008	20.95	0.008	14.05	0.008	445.62	0.008
Numba3	10.75	0.008	432.12	0.222	212.23	0.032	19.59	0.008	10.63	0.008	233.97	0.008
PyPy2	30.64	0.008	24.31	0.008	**177.10**	0.008	**8.73**	0.008	26.78	0.008	**14.56**	0.008
PyPy3	18.09	0.008	**23.58**	0.008	256.08	0.008	8.50	0.008	23.16	0.008	**14.55**	0.008
Shedskin	**8.00**	0.008			380.28	0.008	45.01	0.008	**7.54**	0.008	564.85	0.008

As we want to study the position of these implementations with regards to the references, we proceed with agglomerative *hierarchical cluster analysis* (HCA) [57]. HCA is a method that groups the different implementations based on their energy consumption. This technique is a bottom-up approach as it starts with each implementation being a single cluster, and then merges the two most similar clusters until all the clusters are merged into a single cluster. The similarity between two clusters is measured by the maximum distance between their furthest points. The distance between two points is calculated as the Euclidean distance between the two points. Figure 5.18 displays the Dendogram produced by the HCA.

Except GraalPython, one can notice 3 main clusters. The first cluster contains the reference implementations (CPython2 and CPython3) and the implementations that are based on interpreters (IronPython, IntelPython, ActivePython, Jython, and Nuitka). Moreover, each implementation is the closest to its reference Python version. The interpreters that are based on other virtual machines, such as Jython with JVM and IronPython with .NET, behave slightly differently from the reference implementation. As for the MicroPython, It is the furthest from the references due to his behavior toward exceptions. The second cluster groups the interpreter implementations that are based on a JIT compiler (PyPy2 and PyPy3). Closer to them we find Shedskin, which is a translator that converts the Python code into C++ code

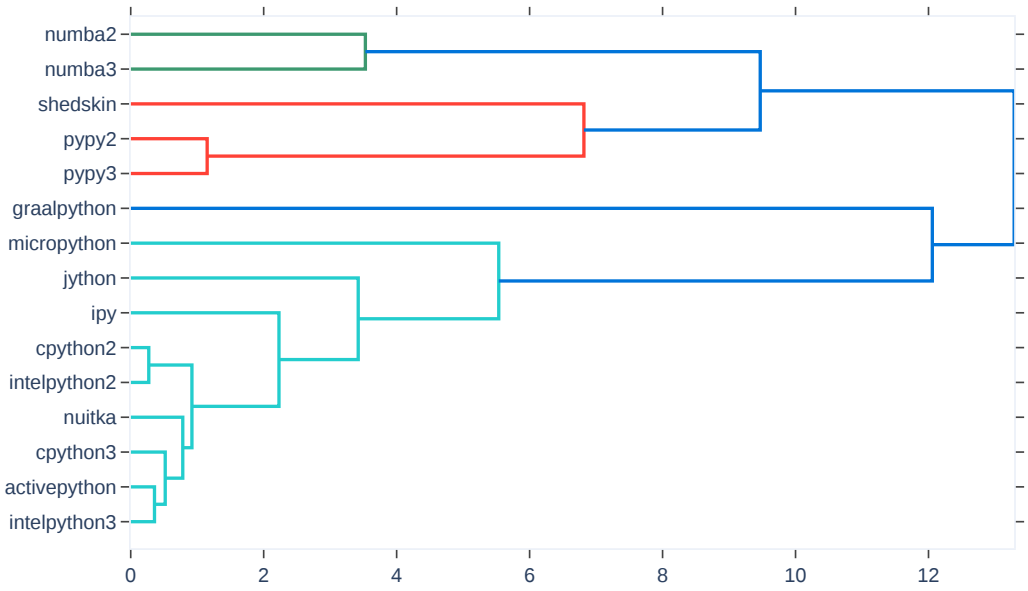


Figure 5.18: Dendogram of the different implementations

to be later compiled into binary code. Table 5.5 shows that this cluster is by far the most energy-efficient one. While Numba2 and Numba3 are also JIT libraries, they differ from PyPy due to their manual optimization. Unlike PyPy, which will decide by itself which part of Python to optimize. In Numba, the developer must target the portion of the code that will be passed to JIT, while the rest of the code will be executed by the regular interpreter (aka CPython).

Remark: Because GraalPython was still in its early stages, some techniques required an abnormally lengthy time, affecting the clustering algorithm.

To help practitioners to choose the best implementation for their use case, we have created a chart that describes the behavior of each implementation toward a specific aspect of programming. Figure 5.19 introduces a radar plot for all the implementations that have been used in this experimentation.

This graph summarizes each implementation energy score when executing our benchmark. The lower the energy consumption, the higher the score. We adopted a logarithmic scale to help the practitioners to compare numerous implementations due to the large differences between them.

As one can notice in Figure 5.19, there is no evolution between CPython2 and CPython3. Moreover, IntelPython, and ActivePython all behave similarly to the reference implementation. Therefore, one can conclude that the work done on those interpreters is primarily to improve a specific purpose and not the core interpreter. ActivePython states that their version is focused on security and prepackaged libraries, which explains why it is slower than other versions due to the addition of this security support.[41] As for IntelPython, it is designed for machine learning. Unfortunately, the TOMMTI benchmark is geared for general-purpose programming. Despite the fact that Nuitka is a compiler, there was no discernible difference in energy consumption. It was even more similar to the reference Python implementation than other interpreters. However, if we look at the Nuitka techniques, we can see that they simply encapsulate the Python code with an interpreter into a single executable. Finally, Shedskin reports on the best energy consumption pattern when it comes to arithmetic operations. One can conclude it is due to the fact of the native type of the variables, unlike in the JIT, where they are treated as objects in the beginning.

Regarding the other VM-based interpereters, Jython and Ipy lacked in terms of energy optimization, which expected as they were at the beginning of the development stage and the main purpose of such implementation is to link the bytecode generated by Jython and IronPython with their respective virtual machines.

[41] https://www.activestate.com/solutions/why-activestate/

Unlike the previous interpeters, GraalPython exhibits a certain promise when it comes to complex algorithms (nested loops in particular).

5.7.5 Python & Binary Operations

The second part of this study is to look at how different Python implementations work when executing different binary operations.

Table 5.20 provides a detailed report on the energy consumption of different implementations of the code. Besides *Cython*, each implementation treats the binary operation the same. As one can see, Table 5.20 shows no difference in the energy consumption of those operations. However, when it comes to Cython, the rotation operations tend to consume 5 times more energy than the other operations, and left rotations consume even more than the right ones. The reason behind such behavior is the fact that Cython does not have a native implementation for these two operations, so it replaces them with more complex operations, which leads to an overhead of 6s for the right rotation and 10s for the left, compared to 1.5s for the other operations. Therefore, it is safe to summarize all the operations into one that we call *binary*.

Figure 5.21 depicts the average energy consumption of different implementations for the binary operations. One can notice 3 clusters. The first, which is the interpreted one, uses approximately 500 Joules per operation. The second is a non-optimized compilation that includes Cython and Nuitka and uses 10 times less energy than the first. And the final one, which has been fully optimized, uses 50 percent less energy. This category includes JIT libraries, also known as Numba, Python implementations that include JIT, also known as PyPy, and Shedskin, which transpile the source code into C and then use a C/C++ compiler to generate the binary, as opposed to Cython and Nuitka, which directly compile the Python code. To conclude this experiment, JIT has a huge impact when it comes to binary operations, and an inadequate runtime selection can lead to a 100x increase in energy consumption. Furthermore, we observed that different binary operations exhibit the same energy footprint for all the implementations, except Cython.

5.7.6 Conclusion

One can conclude that the choice of Python interpreter has a significant impact on the programs' energy consumption. This investigation is made more intriguing by the absence of a universal solution. The primary downside is the incompatibility of some of these solutions, which causes us to make concessions when we need a generic answer.

5.7.7 threads to validity *TODO : missing*

5.8 Conclusion *TODO : missing*


activepython
cpython2
cpython3
graalpython
intelpython2
intelpython3
ipy
jython
micropython
nuitka
numba2
numba3
pypy2
pypy3
shedskin
hashes
except
doubleArithmetic
array
vector
trig
stringConcat
nestedLoop
matrixMultiply
longArithmetic
io
intArithmetic
heapsort
1
10
100
5000
1000

benchmark	Addition	Right Rotation	Left Rotation	Or	XOR	Average
ActivePython	676.980	763.208	651.783	743.016	728.828	712.76
CPython2	441.082	435.886	430.846	415.247	419.081	428.43
cpython3	595.209	685.085	563.839	657.972	655.560	531.53
Cython	35.077	182.688	274.177	34.868	34.504	112.26
Nuitka	33.260	32.980	33.256	33.472	33.030	33.2
Numba2	9.102	8.411	9.460	9.375	9.755	**9.22**
Numba3	9.566	10.144	9.219	9.344	9.665	**9.59**
PyPy2	8.456	7.844	8.286	8.138	7.952	**8.13**
PyPy3	7.552	8.093	8.108	8.669	8.623	**8.21**
Shedskin	8.024	8.070	8.399	8.126	8.277	**8.18**

Figure 5.20: Energy consumption of different implementations using bit operations benchmark (in Joules).

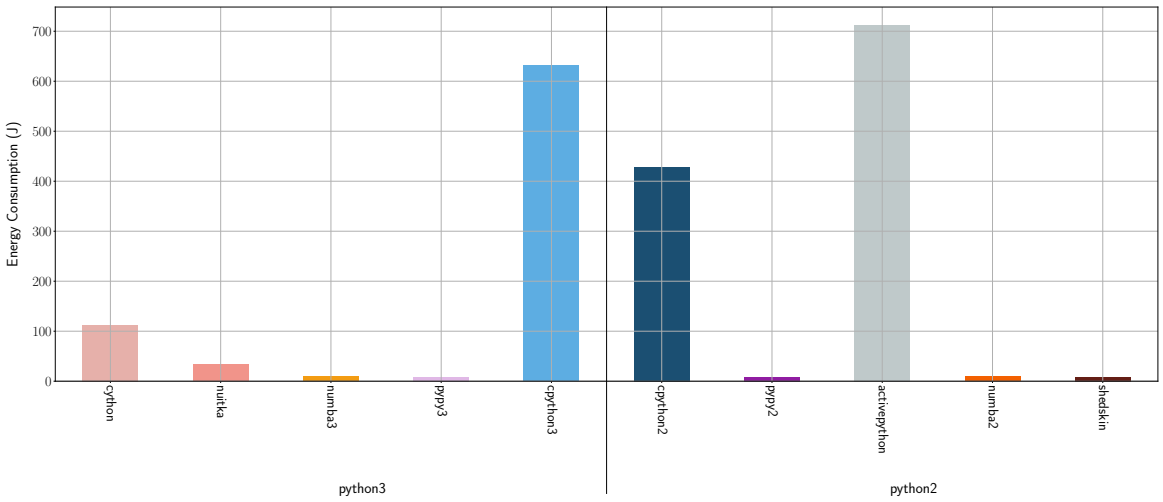


Figure 5.21: Summary of the binary operations for different python implementations

Chapter 6

The Impact of Java Virtual Machine on Energy Consumption

6.1 Introduction

As reported in the state of the art, Java is one of the most popular programming languages adopted by practitioners.In fact, in 2022, Java will be second only to Python, according to PYPY [1]. Furthermore, if we take into consideration legacy applications, Java becomes the most used programming language.According to the TIOBE index, Java was the most frequently used language from 2002 until 2017, and it remained in the top 5 after that [2]. In addition to its popularity, Java exhibits an interesting behavior when it comes to energy consumption and performance, Java applications can be at the same time one of the most energy-efficient or hungry solutions. As we have seen in the previous chapter, an inappropriate combination of parameters can drive Java applications from the top language to the bottom just by setting the wrong parameters. in this chapter, we want to dig deeper into this aspect of Java and study its runtime. This chapter thus focuses on the impact of the runtime of Java applications on energy consumption.

6.1.1 Characteristics of JVM

Java's portability is a core design goal, which means that Java applications will work exactly the same on any operating system and on any hardware.As we see in figure 6.1, Instead of machine code, this is accomplished by compiling Java language code to an intermediate representation known as Java bytecode. Java bytecode instructions are similar to machine

[1] https://pypl.github.io/PYPL.html
[2] https://www.tiobe.com/tiobe-index/

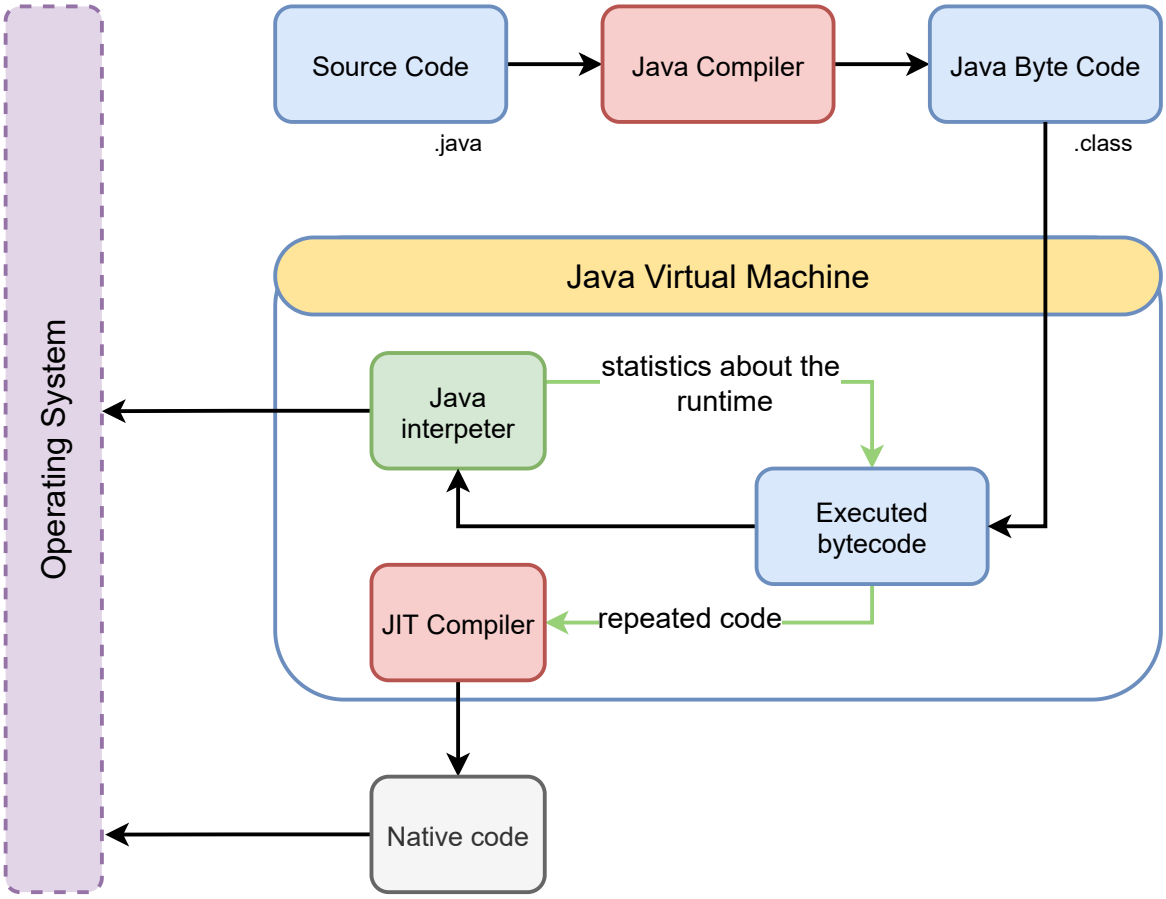


Figure 6.1: JVM architecture

code, but they are executed by a JVM rather than directly on the machine. As a result, Java programs will run slower and consume more memory than C++ programs. However, Just-in-Time (JIT) compilation, which is built into the JVM, improves performance by converting bytecode to machine code at runtime. Furthermore, an automatic garbage collector (GC) is used by Java to manage memory throughout the object lifecycle and to recover memory when objects are no longer in use.

6.1.2 Research questions

In this section, we will investigate the following research questions:

RQ 1: *What is the impact of existing JVM distributions on the energy consumption of Java-based software services?*

RQ 2: *What are the relevant JVM settings that can reduce the energy consumption of a given software service?*

To answer those research questions, we conduct an empirical study to highlight the impact of this runtime.

6.2 Experimental Protocol

To investigate the effect that can have the JVM distribution choice and/or parameters on software energy consumption, we conducted a wide set of experiments on a cluster of machines and used several established Java benchmarks and JVM configurations.

6.2.1 Measurement Contexts

Software Settings. For the sake of reproducibility, each experiment runs within a Docker container based on SDKMAN[3] image and Alpine docker.[4]

Hardware Settings. To report on reproducible measurements, we used the cluster Dahu from the G5K platform [8] for most of our experiments. This cluster is composed of 32 identical compute nodes, which are equipped with 2 Intel Xeon Gold 6130 and 192 GB of RAM. Our experimental protocol enforces that the software under test is the only process executed on the node configured with a very minimal Linux Debian 9 (4.9.0 kernel version). The minimal OS configuration ensures that only mandatory services and daemons are kept active

[3]https://sdkman.io
[4]https://github.com/alpinelinux/docker-alpine

Table 6.1: List of selected JVM distributions.

Distribution	Provider	Support	Selected versions
HOTSPOT	**Adopt OpenJDK**	ALL	8.0.275, 11.0.9, 12.0.2, 13.0.2, 14.0.2, 15.0.1
HOTSPOT	**Oracle**	ALL	8.0.265, 9.0.4, 10.0.2, 11.0.2, 12.0.2, 13.0.2, 14.0.2, 15.0.1, 16.ea.24
ZULU	**Azul Systems**	ALL	8.0.272, 9.0.7, 10.0.2, 11.0.9, 12.0.2, 13.0.5, 14.0.2, 15.0.1
SAPMACHINE	**SAP**	ALL	11.0.9, 12.0.2, 13.0.2, 14.0.2, 15.0.1
LIBRCA	**BellSoft**	ALL	8.0.275, 11.0.9, 12.0.2, 13.0.2, 14.0.2, 15.0.1
CORRETTO	**Amazon**	MJR	8.0.275, 11.0.9, 15.0.1
HOTSPOT	**Trava OpenJDK**	LTS	8.0.232, 11.0.9
DRAGONWELL	**Alibaba**	LTS	8.0.272, 11.0.8
OPENJ9	**Eclipse**	ALL	8.0.275, 11.0.9, 12.0.2, 13.0.2, 14.0.2, 15.0.1
GRAALVM	**Oracle**	LTS	19.3.4.r8, 19.3.4.r11, 20.2.0.r8, 20.2.0.r11
MANDREL	**Redhat**	LTS	20.2.0.0

to conduct robust experiments and reduce the factors that can affect the energy consumption measurements during our experiments [88].

Java Virtual Machines Candidates. We considered a set of 52 JVM distributions taken from 8 different providers/packagers mostly obtained from SDKMAN, as listed in Table 6.1. Depending on providers, either all the versions, majors, or LTS are made available by SDKMAN.

6.2.2 Workload

We ran our experiments across 12 Java benchmarks we picked from OpenBenchmarking.org.[5] This includes 5 acknowledged benchmarks from the DACAPO benchmark suite v. 9.12 [12], namely Avrora, H2, Lusearch, Sunflow and PMD, that have been widely used in previous studies and proven to be accurate for memory management and computer architecture [67, 58]. It consists of open-source and real-world applications with non-trivial memory loads. Then, we also considered 7 additional benchmarks from the RENAISSANCE benchmark suite [96, 97], namely ALS, Dotty, Fj-kmeans, Neo4j, Philosophers, Reaction and Scrabble, which offer a diversified set of benchmarks aimed at testing JIT, GC, profilers, analyzers, and other tools. The benchmarks we picked from both suites exercise a broad range of programming paradigms, including concurrent, parallel, functional, and object-oriented programming. Table 6.2 summarizes the selected benchmarks with a short description. Meanwhile, Figure 6.2 highlights the scope of each benchmark from the test suite.

[5]https://openbenchmarking.org

Table 6.2: List of selected open-source Java benchmarks taken from DACAPO and RENAISSANCE.

Benchmark	Description	Focus
ALS	Factorize a matrix using the alternating least square algorithm on spark	Data-parallel, compute-bound
Avrora	Simulates and analyses for AVR microcontrollers	Fine-grained multi-threading, events queue
Dotty	Uses the dotty Scala compiler to compile a Scala code-base	Data structure, synchroniza-tion
Fj-Kmeans	Runs K-means algorithm using a fork-join framework	Concurrent data structure, task parallel
H2	Simulates an SQL database by executing a TPC-C like benchmark written by Apache	Query processing, transac-tions
Lusearch	Searches keywords over a corpus of data comprising the works of Shakespeare and the King James bible	Externally multi-threaded
Neo4j	Runs analytical queries and transactions on the Neo4j database	Query Processing, Transac-tions
Philosophers	Solves dining philosophers problem	Atomic, guarded blocks
PMD	Analyzes a list of Java classes for a range of source code problems	Internally multi-threaded
Reactors	Runs a set of message-passing workloads based on the reactors framework	Message-passing, critical-sections
Scrabble	Solves a scrabble puzzle using Java streams	Data-parallel, memory-bound
Sunflow	Renders a classic Cornell box; a simple scene com-prising two teapots and two glass spheres within an illuminated box	Compute-bound

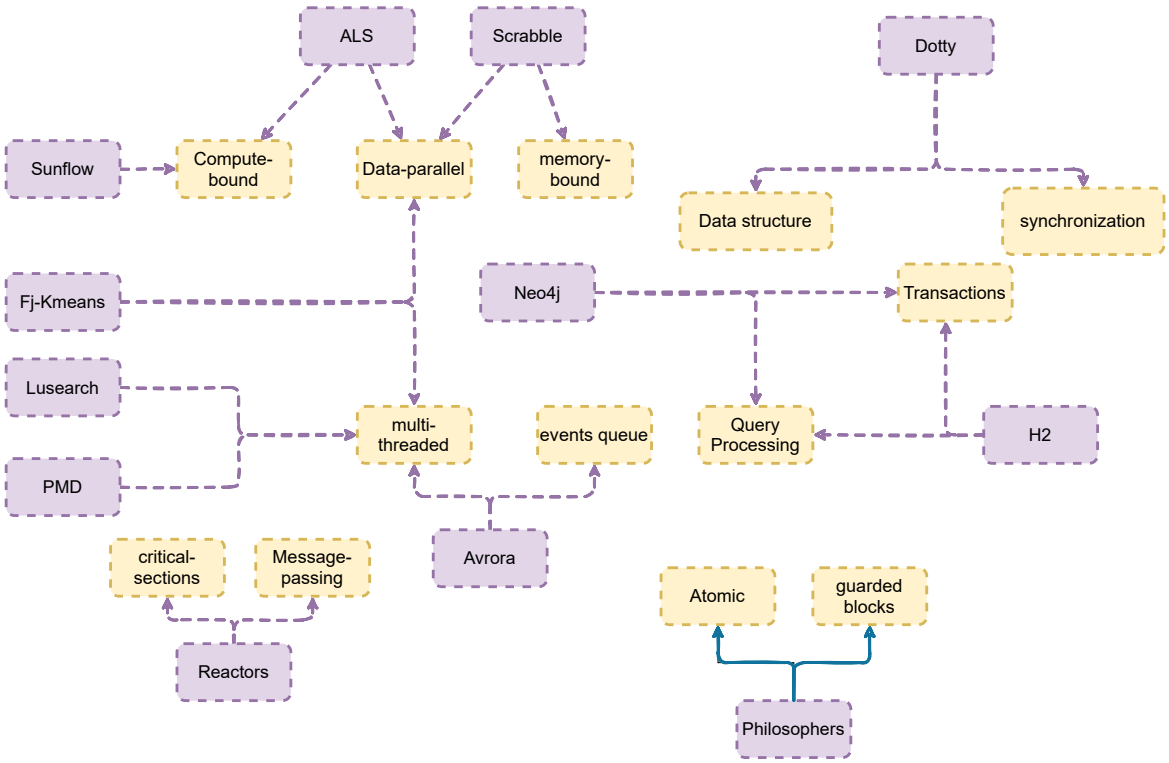


Figure 6.2: Target scope of DACAPO and RENAISSANCE benchmarks.

6.2.3 Metrics and Measurement

Since the goal of this study is the green aspect of JVM, our key metric will be the energy consumed by job completed for each JVM configuration. In addition to the energy consumption, we collected additional metrics to explain the reasons behind the behavior of each experiment. Those additional metrics are:

- execution time,
- number of threads.

Energy Measurements. We used Intel RAPL as a physical power meter to analyze the energy consumption of the CPU package and the DRAM. RAPL is one of the most accurate tools to report on the global energy consumption of a processor [59, 30]. We note that, due to CPU energy consumption variations issues [88], we used the same node for all our experiments. Moreover, we tried to be very careful, while running our experiments, not to fall in the most common benchmarking "crimes" [106]. Every single experiment, therefore, reports on energy metrics obtained from at least 20 executions of 50 iterations per benchmark. All of our experiments are available for use/reproducibility from our anonymous repository.[6]

[6]https://anonymous.4open.science/r/jvm-comparaison-213E/Readme.md

Number of threads. To collect the number of active threads used by the experiment, we use the command `top` and record at fixed intervals.

6.2.4 Extension

We also added an extension to the protocol to allow the user to run the same experiment with different configurations. The package is available in the GitHub repository.[7] To add extra **jvm candidates** for the benchmark applications, we added a new configuration file `jvm.sh` in the root directory of the repository, where we put the name and the version of the jvm to be used. For the **input workload**,the benchmarks should be provided in the `benchmarks` directory. As for extra **metrics**, one can create a new script file that monitors the experiment and record the metrics. We provide some examples, such as `recordpower.sh` to measure the instantaneous power and `recordthreads.sh` to measure the number of active threads during the experiment.

For faster experiments, we propose JREFERRAL[8], an open source tool that automatically compares the JVM energy consumption and recommends the most energy-efficient configuration for a given Java application. We further discuss this tool in Section 7.2.

6.3 Experiments & Results

6.3.1 Energy Impact of JVM Distributions

Job-oriented applications. To answer our first research question, we executed 62,400 experiments by combining the 52 JVM distributions with the 12 Java benchmarks, thus reasoning on 100 energy samples acquired for each of these combinations. Figure 6.3 first depicts the accumulated energy consumption of the 12 Java benchmarks per JVM distribution and major versions (or LTS when unavailable). Concretely, We measure the energy consumption of each of the benchmarks and compute the ratio of energy consumption compared to HOTSPOT-8, which we consider as the baseline in this experiment. Then, we sum the ratios of the 12 benchmarks and depict them as percentages in Figure 6.3.

One can observe that, along with time and versions, the energy efficiency of JVM distributions tends to improve (10% savings), thus demonstrating the benefits of optimizations delivered by the communities. Yet, one can also observe that energy consumption may differ from one distribution to another, thus showing that the choice of a JVM distribution may have a substantial impact on the energy consumption of the deployed software services. For

[7]https://github.com/chakib-belgaid/jvm-comparaison
[8]https://github.com/chakib-belgaid/jreferral

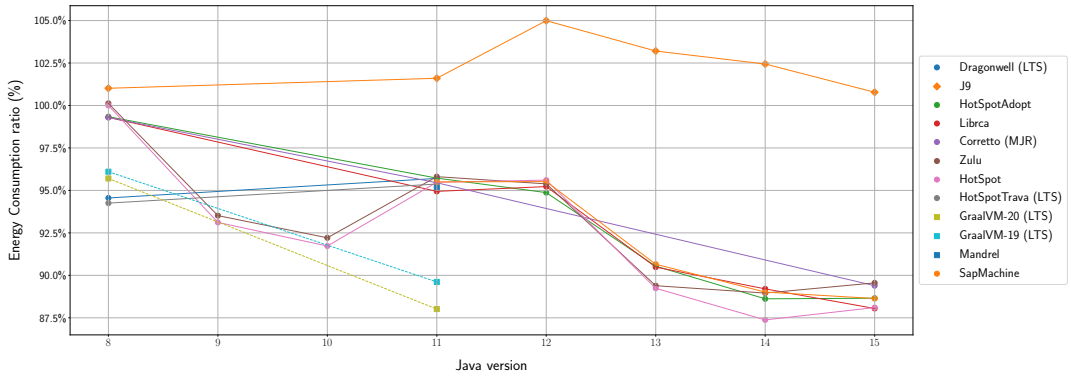


Figure 6.3: Energy consumption evolution of selected JVM distributions along versions.

example, one can note that J9 can exhibit up to 15% of energy consumption overhead, while other distributions seem to converge towards a lower energy footprint for the latest version of Java. As GRAALVM adopts a different strategy focused on LTS support, one can observe that its recent releases provide the best energy efficiency for Java 11, but recent releases of other distributions seem to reach similar efficiency for Java 13 and above, which are recent versions not supported by GRAALVM yet.

Interestingly, this convergence of distributions has been observed since Java 11 and coincides with the adoption of DCE VM by HOTSPOT. Ultimately, 3 clusters of JVMs that encompass JVMs with similar energy consumption can be seen through Figure 6.3: J9, the HOTSPOT and its variants, and GRAALVM. Additional detailed figures to illustrate the evolution of energy consumption per benchmark/JVM are made available from the online repository.[9]

Then, Figure 6.4 depicts the evolution of the energy consumption of the 12 benchmarks, when executed on the HOTSPOT JVM. Figure 6.4 reports on the energy consumption variation of individual benchmarks, using HOTSPOT-8 as the baseline. Our results show that the JVM version can severely impact the energy consumption of the application. However, unlike Figure 6.3, one can observe that, depending on applications, the latest JVM versions can consume less energy (60% less energy for Scrabble) or more energy (25% more energy for the Neo4J). It is worth noticing that the energy consumption of some benchmarks, such as Reactors, exhibit large variations across JVM versions due to experimental features and changes that are not always kept when releasing LTS versions (version 11 here). For example,

[9]https://github.com/chakib-belgaid/jvm-comparaison

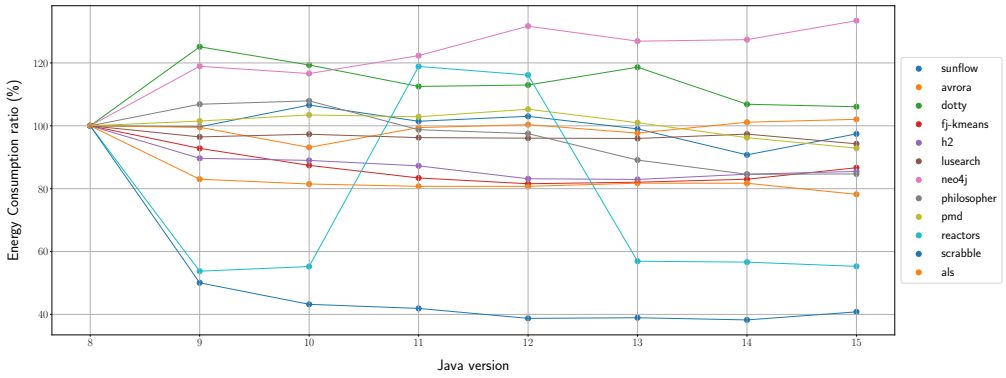


Figure 6.4: Energy consumption of the HotSpot JVM along versions.

the introduction of `VarHandle` to allow low-level access to the memory order modes available in JDK 9 [10] and work along `Unsafe Class` was removed from JVM 11.[11]

Given that the wide set of distributions and versions seems to highlight 3 classes of energy behaviors, the remainder of this chapter considers the following distributions as relevant samples of JVM to be further evaluated: 20.2.0.r11-grl (GRAALVM), 15.0.1-open (HOTSPOT-15), 15.0.21.j9 (J9). We also decided to keep the 8.0.275-open (HOTSPOT-8) as a baseline JVM for some figures to highlight the evolution of energy consumption over time/versions.

Figure 6.5 further explores the comparison of energy efficiency of the JVM distributions per benchmark. One can observe that, depending on the benchmark's focus, the energy efficiency of JVM distributions may strongly vary. When considering individual benchmarks, J9 performs the worst for at least 6 out of 12 benchmarks—*i.e.*, the worst ratio among the 4 tested distributions. Even though, J9 can still exhibit a significant energy saving for some benchmarks, such as Avrora, where it consumes 38% less energy than HOTSPOT and others.

Interestingly, GRAALVM delivers good results overall, being among the distributions with low energy consumption for all benchmarks, except for Reactors and Avrora. Yet, some differences still can be observed with HOTSPOT depending on applications. The newer version of HOTSPOT-15 was averagely good and, compared to HOTSPOT-8, it significantly enhances energy consumption for most scenarios. Finally, Neo4J is the only selected benchmark where HOTSPOT-8 is more energy efficient than HOTSPOT-15.

Service-oriented applications. In this section, instead of considering bounded execution of benchmarks, we run the same benchmarks as services for 20 minutes, and we compare the

[10] https://gee.cs.oswego.edu/dl/html/j9mm.html
[11] https://blogs.oracle.com/javamagazine/the-unsafe-class-unsafe-at-any-speed

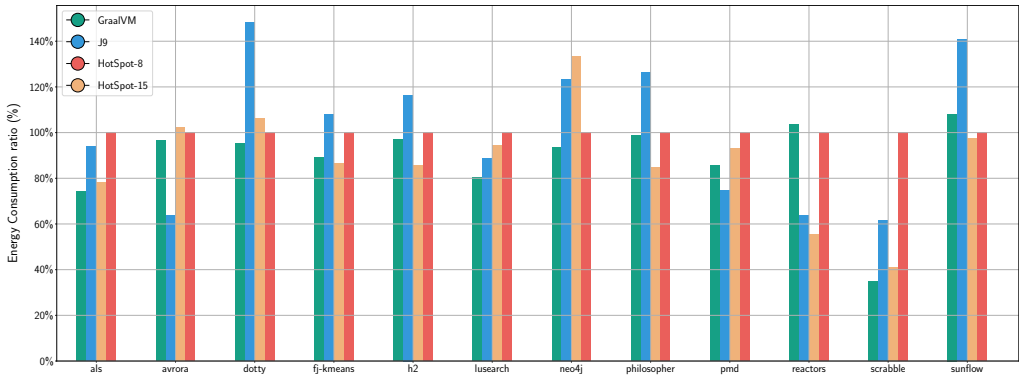


Figure 6.5: Energy consumption comparison across Java benchmarks for HOTSPOT, GRAALVM & J9.

Table 6.3: Power per request for HOTSPOT, GRAALVM & J9.

Benchmark	**JVM**	**Power (P)**	**Requests (R)**	$P/R \times 10^{-3}$
Scrabble	GRAALVM	109 W	5,336 req	20 mW
	HOTSPOT	98 W	3,595 req	27 mW
	J9	92 W	2,603 req	35 mW
Dotty	GRAALVM	45 W	510 req	88 mW
	HOTSPOT	45 W	597 req	75 mW
	J9	46 W	381 req	120 mW

average power and total requests processed by each of the 3 JVM distributions. Globally, the results showed that the average power when using GRAALVM, HOTSPOT, and OPENJ9 is often equivalent and stable over time. This means that the energy efficiency observed for some JVM distributions with job-oriented applications is mainly related to shorter execution times, which incidentally results in energy savings. Nonetheless, we can highlight two interesting observations for two benchmarks whose behaviors differ from others. First, the analysis of the Scrabble benchmark experiments showed that, in some scenarios, some JVMs can exhibit different power consumptions. Figure 6.6 depicts the power consumed by the 3 JVM distributions for the Scrabble benchmark. One can clearly see that GRAALVM requires an average power of 109 W, which is 9 W higher than HOTSPOT-15 and 15 W higher than J9. When it comes to the number of requests processed by Scrabbles during that same amount of time, GRAALVM completes 5,336 requests, against 3,595 for HOTSPOT and 2,603 for J9, as shown in Table 6.3. The higher power usage for GRAALVM helped in achieving a high amount of requests, but also the fastest execution of every request, which was 40% faster on GRAALVM. Thus, GRAALVM was more energy efficient, even if it uses more power, which confirms the results observed in Figure 6.5 for this benchmark.

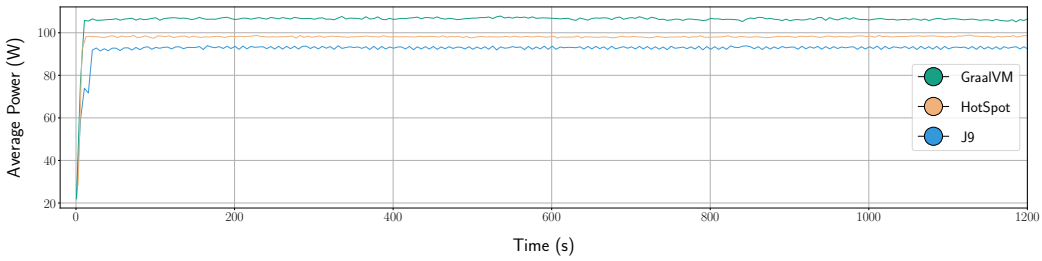


Figure 6.6: Power consumption of Scrabble as a service for HOTSPOT, GRAALVM & J9.

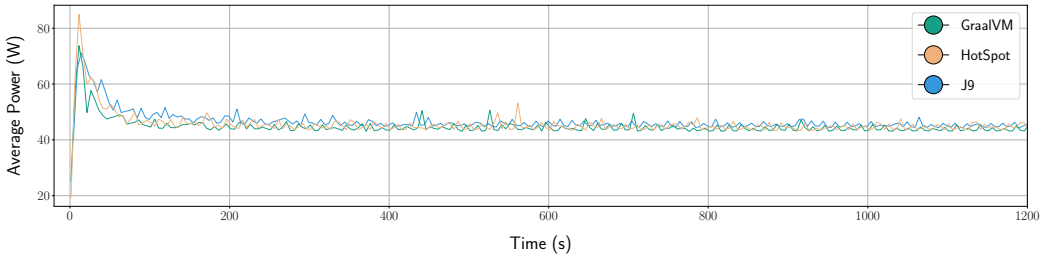


Figure 6.7: Power consumption of Dotty as a service for HOTSPOT, GRAALVM & J9.

The second interesting situation was observed on the Dotty benchmark. More specifically, during the first 100 seconds of the execution of the Dotty benchmark on all evaluated JVMs. At the beginning of the execution, GRAALVM has a slightly lower power consumption, is faster, and consumes 10% less energy. After about 150 seconds, the power differences between the 3 JVMs is barely noticeable. One can, however, notice the effect of the JIT, as HOTSPOT takes the advantage over GRAALVM and becomes more energy efficient. In total, HOTSPOT completes 597 requests against 510 for GRAALVM and 381 for J9, as shown in Table 6.3. HOTSPOT was thus the best choice in the long term, which explains why it is always necessary to consider a warm-up phase and wait for the JIT to be triggered before evaluating the effect of the JVM or the performance of an application. This is precisely what we did in our experiments, and it is why HOTSPOt was more energy efficient than GRAALVM in Figure 6.5; therefore, ignoring the warm-up phase would have been misleading.

To answer **RQ 1**, we conclude that—while most of the JVM platforms perform similarly—we can cluster JVMs in 3 classes: HOTSPOT, J9, and GRAALVM. The choice of one JVM of these classes can have a major impact on software energy consumption, which strongly depends on the application context. When it comes to the JVM version, the latest releases tend to offer the lowest power consumption, but experimental features should be carefully configured, thus further questioning the impact of JVM parameters.

6.3.2 Energy Impact of JVM Settings

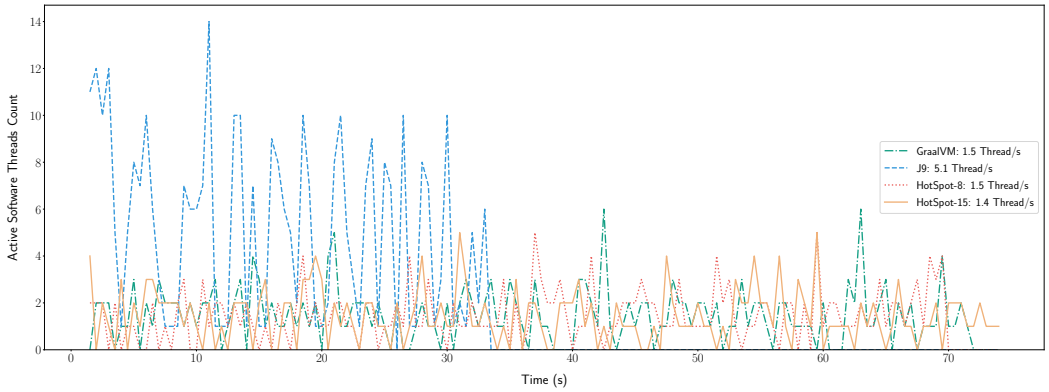
The purpose of our study is not only to investigate the impact of the JVM platform on energy consumption, but also the different JVM parameters and configurations that might have a positive or negative effect, with a focus on 3 available settings: multi-threading, JIT, and GC.

Multithreading

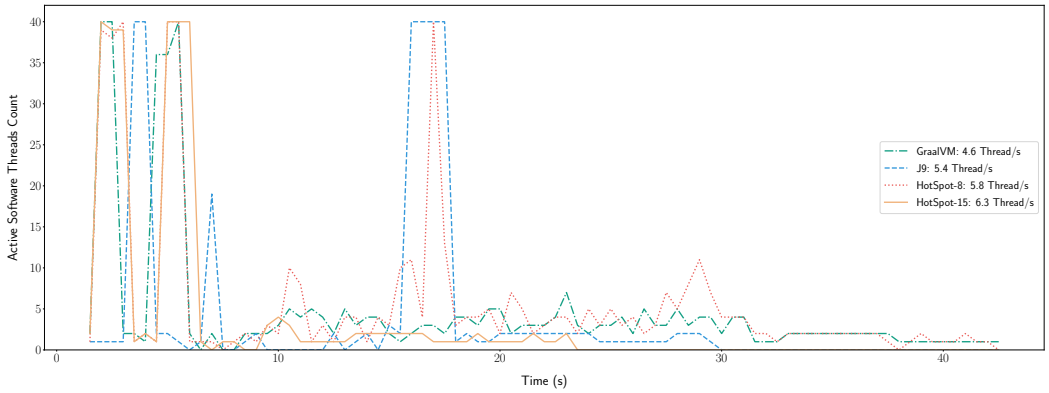
The purpose of this phase is to investigate the impact JVM thread management strategies on energy consumption. This encompasses exploring if the management strategies of application-level parallelism (so-called *threads*) result in different energy efficiencies, depending on JVM distributions.

Investigating such a hypothesis requires a selection of highly parallel and CPU-intensive benchmarks, which is one of the main criteria for our benchmark selection. As no tool can accurately monitor the energy consumption at a thread level, we monitor the global power consumption and CPU utilization during the execution using RAPL for the energy, and several Linux tools for the CPU-utilization (`htop`, `cpufreq`). Knowing that most of the benchmarks are multi-threaded jobs that use multiple cores, further analysis of thread management is required to understand the results of our previous experiments. We thus selected the benchmarks that highlighted the highest differences along JVM distributions from Figure 6.5, namely Avrora and Reactors. We studied their multi-threaded behavior to optimize their energy efficiency.

Figure 6.8 delivers a closer look to the thread allocation strategies adopted by JVM. First, Figure 6.8a illustrates the active threads count evolution over time (excluding the JVM-related threads, usually 1 or 2 extra threads depending on the execution phase) for Avrora. One can notice through the figure that J9 exploits the CPU more intensively by running much more parallel threads compared to other JVMs (an average of 5.1 threads per second for J9 while the other JVMs do not exceed 1.5 thread per second). Furthermore, the number of context switches is twice bigger for J9, while the number of soft page faults is twice smaller. The efficient J9 thread management explains why running the Avrora benchmark took much less time and consumed less energy, given that no other difference for the JIT or GC configuration was spotted between the JVMs. Another key reason for the J9's efficiency for the Avrora benchmark is memory allocation, as OpenJ9 adopts a different policy for the heap allocation. It creates a non-collectible *thread local heap* (TLH) within the main heap for each active thread. The benefit of cloning a dedicated TLH is the fast memory access for independent threads: each thread has its heap and no deadlock can occur.



(a) Active threads of Avrora when using HOTSPOT, GRAALVM, or J9.



(b) Active threads of Reactors when using HOTSPOT, GRAALVM, or J9.

Figure 6.8: Active threads evolution when using HOTSPOT, GRAALVM, or J9.

The second example in Figure 6.8b depicts the active threads evolution over time of the Reactors benchmark. In this case, all the JVMs have a close average of threads per second. Nevertheless, one can still observe that HOTSPOT-15 and J9 keep running faster, which confirms the results of Figure 6.5, where both JVMs consume much less energy compared to GRAALVM and HOTSPOT-8. This difference in energy consumption between benchmarks can be less likely caused by thread management for the Reactors benchmark, as HOTSPOT-8 reports on a higher average of active threads. However, the TLH mechanism was not as efficient as for the Avrora benchmark, as dedicating a heap for each thread can also cause some extra memory usage for data duplication and synchronization, especially if a lot of data is shared between threads.

In conclusion, JVMs thread management can sometimes constitute a key factor that impacts software energy consumption. However, we suggest checking and comparing JVMs before deploying a software, especially if the target application is parallel and multi-threaded.

Just-in-Time Compilation

The purpose of experiments on JIT is to highlight the different strategies that can impact software energy consumption within a JVM and between JVMs. We identified a set of JIT compiler parameters for every JVM platform.

For J9, we considered fixing the intensity of the JIT compiler at multiple levels (cold, warm, hot, veryhot, and scorching).[12] The hotter the JIT, the more code optimization to be triggered. We also varied the minimum count method calls before a JIT compilation occurs (10, 50, 100), and the number of JIT instances threads (from 1 to 7). For HOTSPOT-15, we conducted experiments while disabling the tiered compilation (that generates compiled versions of methods that collect profiling information about themselves), and we also varied the JIT maximum compilation level from 0 to 4, we also tried out HOTSPOT with a basic GRAALVM JIT. We note that level 0 of JIT compilation only uses the interpreter, with no real JIT compilation. Levels 1, 2, and 3 use the C1 compiler (called client-side) with different amounts of extra tuning. The JIT C2 (also called server-side JIT) compiler only kicks in at level 4.

For GRAALVM, we conducted experiments with and without the JVMCI (a Java-based JVM compiler interface enabling a compiler written in Java to be used by the JVM as a dynamic compiler). We also considered both the community and economy configurations (no enterprise). A JIT+AOT (*Ahead Of Time*) disabling experiment has also been considered for all of the 3 JVM platforms. Table 6.4 reports on the energy consumption of the experiments we conducted for most of the benchmarks and JIT configurations under study.

[12][https://www.eclipse.org/openj9/docs/jit/]

Table 6.4: Energy consumption when tuning JIT settings on HOTSPOT, GRAALVM & J9

JVM	Mode	ALS		Avrora		Dotty		Fj-kmeans		H2	
	Default	2848	*p-values*	3861	*p-values*	2271	*p-values*	948	*p-values*	1959	*p-values*
GRAALVM	DisableJVMCI	3099	**0.001**	4012	**0.041**	2694	**0.001**	934	**0.011**	1771	**0.005**
	Economy	4503	**0.001**	3895	0.793	3466	**0.001**	1306	**0.002**	2560	**0.001**
	Default	3792	*p-values*	2122	*p-values*	3515	*p-values*	1271	*p-values*	2426	*p-values*
	Thread 1	4157	**0.001**	2121	0.875	4749	**0.001**	1297	0.097	2597	0.066
	Thread 3	3849	0.018	2105	0.713	3574	0.104	1259	0.371	2450	0.637
	Thread 7	3843	0.041	2386	0.372	3511	0.875	1259	0.25	2424	0.637
	Count 0	8461	**0.001**	2425	**0.001**	4877	**0.001**	2289	**0.002**	3212	**0.001**
J9	Count 1	4281	**0.001**	2150	0.431	3164	**0.001**	1841	**0.002**	2546	0.431
	Count 10	3980	**0.001**	2431	0.713	3771	**0.001**	1312	**0.011**	2779	**0.003**
	Count 100	3878	**0.007**	2141	0.713	3469	0.227	1363	0.523	2513	0.128
	Cold	6788	**0.001**	2134	0.637	4855	**0.001**	1636	**0.002**	2873	**0.001**
	Warm	4594	**0.001**	2112	0.713	4253	**0.001**	1244	0.055	2521	0.128
	Hot	7553	**0.001**	2310	**0.001**	12749	**0.001**	1452	**0.002**	3973	**0.001**
	VeryHot	15113	**0.001**	3300	**0.001**	18235	**0.001**	2430	**0.002**	7205	**0.001**
	Schorching	18316	**0.001**	3541	**0.001**	21686	**0.001**	2514	**0.002**	7855	**0.001**
	Default	2997	*p-values*	4014	*p-values*	2516	*p-values*	934	*p-values*	1796	*p-values*
	Graal	2999	0.637	3971	0.318	2512	0.318	929	0.609	1662	**0.007**
HOTSPOT	Lvl 0	491443	/	14484	/	84395	/	/	/	52344	/
	Lvl 1	/	/	3731	**0.001**	3302	**0.001**	1256	**0.002**	2523	**0.001**
	Lvl 2	3079	**0.004**	4110	0.189	3723	**0.001**	22547	**0.002**	2840	**0.001**
	Lvl 3	16375	**0.001**	7729	**0.001**	6789	**0.001**	144914	**0.002**	4139	**0.001**
	NotTired	3254	**0.001**	3901	0.189	3110	**0.001**	912	**0.021**	1846	0.227

JVM	Mode	Neo4j		Pmd		Reactors		Scrabble		Sunflow	
	Default	3313	*p-values*	297	*p-values*	23452	*p-values*	452	*p-values*	335	*p-values*
GRAALVM	DisableJVMCI	5086	**0.001**	353	**0.001**	25007	**0.007**	503	**0.002**	354	0.227
	Economy	9525	**0.001**	270	**0.001**	30317	**0.001**	649	**0.002**	392	**0.002**
	Default	4336	*p-values*	277	*p-values*	12705	*p-values*	734	*p-values*	476	*p-values*
	Thread 1	4906	**0.001**	350	**0.001**	12800	0.713	948	**0.002**	626	**0.005**
	Thread 3	4477	**0.005**	294	**0.004**	12647	0.875	795	0.021	457	0.27
	Thread 7	4431	0.104	273	0.372	12600	0.875	808	0.055	463	0.372
	Count 0	10565	**0.001**	744	**0.001**	18084	**0.001**	1476	**0.002**	922	**0.001**
J9	Count 1	7166	**0.001**	272	0.128	14715	**0.001**	1005	**0.002**	514	0.052
	Count 10	4979	**0.001**	299	**0.001**	12000	0.104	860	**0.005**	1182	**0.001**
	Count 100	4547	**0.001**	262	0.031	12313	0.024	768	0.16	634	**0.004**
	Cold	7250	**0.001**	275	0.372	20380	**0.001**	870	**0.005**	386	**0.001**
	Warm	5305	**0.001**	411	**0.001**	13726	**0.001**	913	**0.002**	336	**0.001**
	Hot	8979	**0.001**	857	**0.001**	36534	**0.001**	1180	**0.002**	506	0.128
	VeryHot	19359	**0.001**	793	**0.001**	38303	**0.001**	5420	**0.002**	1692	**0.001**
	Schorching	26409	**0.014**	808	**0.001**	43929	**0.001**	5583	**0.002**	1778	**0.001**
	Default	4787	*p-values*	323	*p-values*	11685	*p-values*	530	*p-values*	325	*p-values*
	Graal	4750	0.372	327	0.189	11548	0.523	537	0.701	338	0.564
HOTSPOT	Lvl 0	356287	/	1073	/	148381	/	/	/	14559	/
	Lvl 1	8304	**0.001**	222	**0.001**	22410	**0.002**	735	**0.002**	277	**0.007**
	Lvl 2	19058	**0.001**	226	**0.001**	40701	**0.002**	2291	**0.002**	4131	**0.001**
	Lvl 3	44594	**0.001**	330	**0.005**	190124	**0.002**	9070	**0.002**	10449	**0.001**
	NotTired	3844	**0.001**	933	**0.001**	11256	**0.041**	588	**0.003**	405	**0.001**

The *p*-values are computed with the Mann-Whitney test, with a null hypothesis of the energy consumption being equal to the default configuration. The *p*-values in bold show the values that are significantly different from the default configuration with a 95% confidence, where the values in green highlight the strategies that consumed significantly less energy than the default (less energy and significant *p*-value).

For J9, we noticed that adopting the default JIT configuration is always better than specifying a custom JIT intensity. The warm configuration delivers the closest results to the best results observed with the default configuration. Moreover, choosing a low minimum count of method calls seems to have a negative effect on the execution time and energy consumption. The only parameter that can give better performance than the default configuration in some cases is the number of parallel JIT threads—using 3 and 7 parallel threads—but is not statistically significant.

For GRAALVM, the default community configuration is often the one that consumes the least energy. Disabling the JVMCI can—in some cases—have a benefit (16% of energy consumption reduction for the H2 benchmark), but still gave overall worst results (80% more energy consumption for the Neo4J benchmark). In addition, switching the economy version of the GRAALVM JIT often results in consuming more energy and delaying the execution.

For HOTSPOT, keeping the default configuration of the JIT is also mostly good. The usage of the C2 JIT is often beneficial (JIT level 4) in most cases while using the GRAALVM JIT reported similar energy efficiency. Yet, some benchmarks showed that using only the C1 JIT (JIT level 1) is more efficient and even outperforms the usage of the C2 compiler. 10% on Avrora and 30% on Pmd are examples of energy savings observed by using the C1 compiler. However, being limited to the C1 compiler can also cause a huge degradation in energy consumption, such as 32% and 34% of additional energy consumed for the Dotty and FJ-kmeans benchmarks, respectively. Hence, if it is a matter of not using the C2 JIT, the experiments have shown that the level 1 JIT is always the best, compared to levels 2 or 3 that also use the C1 JIT, but with more options, such as code profiling that impacts negatively the performance and the energy efficiency. Level 0 JIT compilation should never be an option to consider. No *p*-value has been computed for Level 0, due to the limited amount of iterations executed with this mode (very high execution time, clearly much more consumed energy).

Globally, we conclude through these experiments that keeping the default JIT configuration was more energy efficient in 80% of our experiments and for the 3 classes of JVMs. This advocates using the default JIT configuration that can often deliver near-optimal energy efficiency. Although, some other configurations, such as using only the C1 JIT or disabling the JVMCI could be advantageous in some cases.

Table 6.5: The different J9 GC policies

Policy	**Description**
Balanced	Evens out pause times & reduces the overhead of the costlier operations associated with GC
Metronome	GC occurs in small interruptible steps to avoid stop-the-world pauses
Nogc	Handles only memory allocation & heap expansion, with no memory reclaim
Gencon (default)	Minimizes GC pause times without compromising throughput, best for short-lived objects
Concurrent Scavenge	Minimizes the time spent in stop-the-world pauses by collecting nursery garbage in parallel with running application threads
optthruput	Optimized for throughput, stopping applications for long pauses while GC takes place
Optavgpause	Sacrifices performance throughput to reduce pause times compared to optthruput

Table 6.6: The different HOTSPOT/GRAALVM GC policies

Policy	**Description**
G1GC (default)	Uses concurrent & parallel phases to achieve low-pauses GC and maintain good throughput
SerialGC	Uses a single thread to perform all garbage collection work (no threads communication overhead)
ParallelGC	Known as throughput collector: similar to SerialGC, but uses multiple threads to speed up garbage collections for scavenges
parallelOldGC	Use parallel garbage collection for the full collections, enabling it automatically enables the ParallelGC

Garbage Collection

Changing or tuning the GC strategy has been acknowledged to impact the JVM performances [68]. To investigate if this impact also benefits energy consumption, we conducted a set of experiments on the selected JVMs. We considered different garbage collector strategies with a limited memory quantity of 2 GB, and recorded the execution time and the energy consumption. The tested GC strategies options mainly vary between J9 and the other 2 JVMs, as detailed in Table 6.5.

For HOTSPOT and GRAALVM, we also considered many GC policies, as described in Table 6.6. Furthermore, other GC settings have also been tested for all JVM platforms, such as the *pause time*, the *number of parallel threads* and *concurrent threads* and *tenure age*.

Table 6.7 summarizes the results of all the tested GC strategies with our selected benchmarks and the p-values of the Mann-Whitney test, with a null hypothesis of the energy consumption being equal to the default configuration with a 95% confidence. The p-values in

Table 6.7: Energy consumption when tuning GC settings on HOTSPOT, GRAALVM & J9

JVM	Mode	ALS		Avrora		Dotty		H2		Neo4j	
	Default	2570	*p-values*	4153	*p-values*	2223	*p-values*	1870	*p-values*	5256	*p-values*
	1Concurent	2567	0.403	4007	**0.023**	2220	1.000	1883	0.982	5368	1.000
	1Parallel	2668	**0.012**	3904	**0.008**	2228	0.835	2022	**0.000**	5836	**0.012**
	5Concurent	2570	0.676	4117	0.161	2215	0.210	1862	0.505	5259	1.000
GRAALVM	5Parallel	2561	0.676	3863	**0.012**	2237	1.000	1910	0.103	5223	0.403
	DisableExplicitGC	2559	0.210	3911	**0.003**	2215	1.000	1978	**0.018**	5106	0.210
	ParallelCG	2720	**0.012**	4016	0.206	2237	0.531	1945	**0.000**	13172	**0.037**
	ParallelOldGC	2715	**0.012**	4032	0.103	2221	1.000	1925	**0.002**	13362	/
	Default	3371	*p-values*	2243	*p-values*	3237	*p-values*	2107	*p-values*	6277	*p-values*
	Balanced	9012	**0.012**	2232	0.597	3429	**0.012**	2247	**0.002**	8853	**0.012**
	ConcurrentScavenge	3487	**0.012**	2270	0.280	3388	**0.012**	2319	**0.001**	6857	**0.012**
	Metronome	2098	**0.012**	2265	0.505	3815	**0.012**	2717	**0.000**	12103	**0.012**
J9	Nogc	3454	**0.022**	2239	0.872	3259	0.144	2207	0.031	61781	**0.012**
	Optavgpause	3601	**0.012**	2431	0.370	3425	**0.012**	2169	0.297	7495	**0.012**
	Optthruput	3357	1.000	2432	0.241	3178	0.403	2194	0.139	6324	0.835
	ScvNoAdaptiveTenure	3494	**0.012**	2253	0.800	3248	0.835	2161	0.103	8442	**0.012**
	Default	2765	*p-values*	4115	*p-values*	2492	*p-values*	1673	*p-values*	8152	*p-values*
	1Concurent	2775	0.060	4137	0.346	2493	0.676	1675	0.918	8062	0.531
	1Parallel	2863	**0.012**	4142	0.800	2526	**0.037**	1853	**0.001**	8270	0.676
	5Concurent	2758	0.676	4091	0.872	2485	0.296	1681	0.608	8087	0.835
	5Parallel	2767	0.144	4176	0.077	2473	0.060	1654	0.720	8046	0.835
HOTSPOT	DisableExplicitGC	2734	**0.012**	4062	0.448	2483	0.835	1702	0.248	7710	**0.037**
	ParallelCG	2653	**0.012**	4064	0.629	2356	**0.012**	1602	**0.008**	8953	0.060
	ParallelOldGC	2764	0.531	4070	0.872	2525	0.802	1675	0.959	7963	0.403
	SerialGC	2593	**0.012**	4083	0.395	2378	**0.012**	1620	**0.046**	5745	**0.012**

JVM	Mode	Pmd		Reactors		Scrabble		Sunflow	
	Default	281	*p-values*	2611	*p-values*	410	*p-values*	353	*p-values*
	1Concurent	286	0.182	2664	1.000	413	0.885	347	0.573
	1Parallel	298	**0.000**	2869	0.144	561	**0.030**	317	**0.000**
	5Concurent	282	0.980	2611	0.531	414	0.885	362	0.356
GRAALVM	5Parallel	282	0.538	2682	0.531	424	0.112	353	0.758
	DisableExplicitGC	281	0.758	2704	0.676	400	0.312	332	**0.036**
	ParallelCG	282	0.878	2267	**0.022**	545	**0.030**	329	**0.003**
	ParallelOldGC	282	0.918	2514	**0.012**	535	**0.030**	329	**0.008**
	Default	232	*p-values*	1644	*p-values*	589	*p-values*	510	*p-values*
	Balanced	235	0.412	1902	**0.020**	661	0.061	519	0.505
	ConcurrentScavenge	233	0.878	1705	0.903	639	0.194	546	**0.018**
	Metronome	239	**0.022**	2089	**0.020**	758	**0.030**	422	**0.000**
J9	Nogc	227	0.151	1505	**0.066**	711	**0.030**	499	0.720
	Optavgpause	253	**0.000**	1772	0.391	1089	**0.030**	478	**0.046**
	Optthruput	232	0.878	1554	0.111	640	0.194	429	**0.000**
	ScvNoAdaptiveTenure	228	0.137	1908	**0.020**	618	0.665	528	0.218
	Default	316	*p-values*	1546	*p-values*	484	*p-values*	347	*p-values*
	1Concurent	316	0.383	1533	0.665	478	0.470	334	**0.218**
	1Parallel	334	**0.000**	1747	**0.030**	592	**0.030**	320	**0.002**
	5Concurent	314	0.330	1497	0.665	469	**0.030**	336	0.259
	5Parallel	316	0.573	1546	0.470	489	0.470	342	0.573
HOTSPOT	DisableExplicitGC	312	0.200	1545	0.470	470	0.061	325	**0.014**
	ParallelCG	300	**0.000**	1476	0.885	579	**0.030**	336	0.081
	ParallelOldGC	314	0.720	1582	0.194	475	0.470	333	0.151
	SerialGC	307	**0.002**	1672	0.061	601	**0.030**	352	0.473

bold show the values that are significantly different from the default configuration, whereas the values in green highlight the strategies that consumed significantly less energy than the default. For GRAALVM, one can see that the GC default configuration is efficient in most experiments, compared to other strategies. The main noticeable impact is related to the ParallelGC and ParallelOldGC. In fact, the ParallelGC can be 13% more energy efficient in some applications with a significant *p*-value, such as Reactors, compared to default. However, the same GC strategy can cause the software to consume twice times more, as for the Neo4j benchmark, due to the high communications between the GC threads, and the fragmentation of the memory.

For J9, the default Gencon GC causes the software to report an overall good energy efficiency among the tested benchmarks. However, other GC can cause better or worse energy consumption than Gencon depending on workloads. Using the Metronome GC consumes 35% less energy for the ALS benchmark and 17% less energy for the Sunflow benchmark, but it also consumes 100% more energy for the Neo4j benchmark and 28% more energy for Reactors. The reason is that Metronome occurs in small preemptible steps to reduce the GC cycles composed of many GC quanta. This suits well for real-time applications and can be very beneficial when long GC pauses are not desired, as observed for ALS. However, if the heap space is insufficient after a GC cycle, another cycle will be triggered with the same ID. As Metronome supports class unloading in the standard way, there might be pause time outliers during GC activities, inducing a negative impact on the Neo4j execution time and energy consumption.

The same goes for the Balanced GC that tries to reduce the maximum pause time on the heap by dividing it into individually managed regions. The Balanced strategy is preferred to reduce the pause times that are caused by global GC, but can also be disadvantageous due to the separate management of the heap regions, such as for ALS where it consumed about three times the energy consumption, compared to the default Gencon GC. On the other hand, the Optthruput GC, which stops the application longer and less frequently, gave very good overall results and sometimes even outperformed the Gencon GC by a small margin. Other JVM parameters, such as the ConcurrentScavenge or noAdaptiveTenure did not have a substantial impact during our experiments.

Finally, the results of HOTSPOT shared similarities with GRAALVM. The ParallelGC happened to give better (6% for Dotty) or worst (10% for Neo4j) energy efficiency compared to the default GC. On the other hand, ParallelOldGC and Serial GC gave better results than the default G1 GC. More specifically, the second one consumed 30% and 6% less energy than the default GC for the Neo4j and Dotty benchmarks, respectively. The most interesting result for HOTSPOT is the 30% energy reduction obtained with the Serial GC. This last was

also more efficient on ALS (6% less energy), compared to the default G1 GC, due to its single-threaded GC that only uses one CPU core.

Unfortunately, we cannot convey predictive patterns on how to configure the GC to optimize energy efficiency. However, some considerations should be taken into account when choosing the GC, such as the garbage collection time, the throughput, etc. Other settings are less trivial to determine, such as tenure age, memory size, and GC threads count. Experiments should thus be conducted on the software to tune the most convenient GC configuration to achieve better energy efficiency in production.

Therefore, we noticed during our experiments that, even if using the default GC configuration ensures an overall steady and correct energy consumption, we still found other settings that reduce that energy consumption in 50% of our experiments. Tuning the GC according to the hosted app/benchmark is thus critical to reducing energy consumption.

To answer **RQ 2**, we conclude that users should be careful while choosing and configuring the garbage collector as substantial energy enhancements can be recorded from one configuration to another. The default GC consumes more energy than other strategies in most situations. However, keeping the default JIT parameters often delivers near-optimal energy efficiency. In addition, the JVM platforms can handle differently multi-threaded applications and thus consume a different amount of time/energy. Dedicated performance tuning evaluations should therefore be conducted on such software to identify the most energy-efficient platform and settings.

6.4 Threats to Validity

Several issues may affect the validity of our work. First, we have the use of the Intel RAPL, one of the most accurate available tools to measure the energy consumption of software [59, 30]. However, RAPL only gives the global energy consumption and no fine-grained measures at process or thread levels. We used bare-metal hardware with a minimal OS and turned off all the non-essential services and daemons to limit the overhead that the OS may add to the execution, even if it is not substantial [88].

Another measurement issue is the CPU energy variation within machines (cf. Chapter 3), thus we executed all the comparable tests on the same node and with the recommended settings to mitigate this threat.

Benchmarks' execution time could also constitute a more subtle threat to the validity of our work, especially for some benchmarks that run fast, such as the Pmd benchmark. We thus gave a lot of attention to how long the benchmark is running for the hardware we used,

Table 6.8: J-Referral recommendations.

Project	Metric	Energy	JVM	Execution flags
Zip4J	Least energy	2210 J	16-sapmchn	default
	Most energy	3680 J	8.0.292-J9	default
K-nucl	Least energy	1296 J	21.1.r16-grl	default
	Most energy	4433 J	15.0.1-J9	-Xjit:optlevel=cold

and we tuned the input data workloads to execute benchmarks for at least many (from 10 to hundreds) seconds. Experiments ran at least 30 times to compute the average consumption and the associated standard deviation, therefore reasoning over reasonable dispersion around the average.

How generalizable are our results? We believe that our study conclusions and guideline remain empirical, as we do not intend to generalize any result we obtained for some JVM or benchmark. We provide practitioners with some prerequisites to check before software deployment to reduce the software energy footprint by considering the JVM and its settings.

6.5 Tools and contributions

J-Referral is an open-source tool designed to assist developers and practitioners in selecting the most energy-efficient JVM configuration for their software.. Table 6.8 illustrates an example of the final report returned by J-Referral. The tool was tested for 2 Java projects: Zip4J[13] and K-nucleotide.[14] Zip4J runs a large file compression, while K-nucleotide extracts a DNA sequence, and updates a hashtable of k-nucleotide keys to count specific values. The short report presented in Table 6.8 shows the ratio of potential energy saving between the most and least energy consuming tested JVM (40% and 70% energy savings for Zip4J and K-nucleotide, respectively). Options are available for J-Referral to obtain much more detailed reports including execution time, DRAM usage, split DRAM vs. CPU consumption, etc. The tool is available as *open-source software* (OSS) from our GitHub repository.[15]

6.6 Conclusion

The results of our investigations show that many JVMs share energy efficiencies and can be grouped into 3 classes: HOTSPOT, J9, and GRAALVM. The 3 selected JVM classes

[13]https://github.com/srikanth-lingala/zip4j
[14]https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/knucleotide.html
[15]https://github.com/chakib-belgaid/jreferral

can however report a different energy efficiency for different software and/or workloads, sometimes by a large margin. While we did not observe a unique champion when it comes to energy consumption, GRAALVM reported the best energy efficiency for a majority of benchmarks. Nonetheless, each JVM can achieve the best or the worst depending on the hosted application. One cause can be thread management strategies, as observed with J9 when advantageously running Avrora. Moreover, some JVM settings can cause energy consumption variations. Our experiments showed that the default JIT compiler of the JVM is often near-optimal, in at least 80% of our experiments. The default GC, however, was outperforming alternative strategies in half of our experiments, with some large gains observed when using some alternative GC depending on the application characteristics.

Our main conclusions and guidelines can be thus summarized as: *i)* testing software on the 3 classes of JVM and identifying the one that consumes the least is a good practice, especially for multi-threading purposes, *ii)* while the JVM default JIT give often good energy consumption results, some settings may improve the energy consumption and could be tested, *iii)* the choice of the GC may lead to a large impact on the energy consumption in many situations, thus encouraging a careful tuning of this parameter before deployment. To ease the integration of the above guidelines, we propose a tool, named J-Referral, to recommend the most energy-efficient JVM distribution and configuration among more than a hundred considered possibilities. It establishes a full report on the energy consumption of both CPU and DRAM components for each JVM distribution and/or configuration to help the user to choose the one with the least consumption for Java software.

Chapter 7

Discussion and Conclusion

7.1 Conclusion*TODO : missing*

7.2 Summary of Contributions*TODO : missing*

This section will describe the contributions of this thesis. These can be summarized as follows:

1. **First Idea:** We proposed ...
2. **Second Idea:** We investigated ...
3. **Third Idea:** We addressed ...

7.3 Limitations and Challenges*TODO : missing*

7.4 Future Work*TODO : missing*

.... Some potential areas for future efforts could include the following:

1. ...
2. ...
3. ...

Bibliography

[1] (2008). *Pearson's Correlation Coefficient*. Springer Netherlands.

[2] (2018). PYPL PopularitY of Programming Language index. https://pypl.github.io/PYPL.html.

[3] Abuabdo, A. and Al-Sharif, Z. A. (2019). Virtualization vs. Containerization: Towards a Multithreaded Performance Evaluation Approach. In *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–6.

[4] Acun, B., Miller, P., and Kale, L. V. (2016). Variation Among Processors Under Turbo Boost in HPC Systems.

[5] Akeret, J., Gamper, L., Amara, A., and Refregier, A. (2015). HOPE: A Python just-in-time compiler for astrophysical computations. *Astronomy and Computing*, 10:1–8.

[6] Avgerinou, M., Bertoldi, P., and Castellazzi, L. (2017). Trends in Data Centre Energy Consumption under the European Code of Conduct for Data Centre Energy Efficiency. *Energies*, 10(10):1470.

[7] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The nas parallel benchmarks—summary and preliminary results.

[8] Balouek, D., Carpen Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez, C., Quesnel, F., Rohr, C., and Sarzyniec, L. (2013). Adding virtualization capabilities to the Grid'5000 testbed. In *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*. Springer.

[9] Bedard, D., Lim, M. Y., Fowler, R., and Porterfield, A. (2010). Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, pages 479–484. IEEE.

[10] Benmoussa, K., Laaziri, M., Khoulji, S., Larbi, K. M., and Yamami, A. E. (2019). A new model for the selection of web development frameworks: Application to PHP frameworks. *International Journal of Electrical and Computer Engineering (IJECE)*, 9(1):695–703.

[11] Blackburn, S. M., Diwan, A., Hauswirth, M., Sweeney, P. F., Nelson Amaral, J., Tuma, P., Pankratius, V., Nystrom, N., Moret, P., Kalibera, T., and et al. (2012). Evaluate collaboratory technical report: Can you trust your experimental results?

[12] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006a). The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA. ACM Press.

[13] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., et al. (2006b). The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190.

[14] Borkar, S. (2005). Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6).

[15] Bourdon, A., Noureddine, A., Rouvoy, R., and Seinturier, L. (2013). PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level. *ERCIM News*, 92.

[16] Breitbart, J., Weidendorfer, J., and Trinitis, C. (2015). Case study on co-scheduling for hpc applications. In *2015 44th International Conference on Parallel Processing Workshops*, pages 277–285. IEEE.

[17] Brown, A. W., Kaiser, K. A., and Allison, D. B. (2018). Issues with data and analyses: Errors, underlying themes, and potential solutions. *Proceedings of the National Academy of Sciences*, 115(11):2563–2570.

[18] Bujnowski, G. and Smołka, J. (2020). Java and kotlin code performance in selected web frameworks. *Journal of Computer Sciences Institute*, 16:219–226.

[19] Bukh, P. N. D. (1992). The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling.

[20] Burtscher, M., Zecena, I., and Zong, Z. (2014). Measuring gpu power with the k20 built-in sensor. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, pages 28–36.

[Caldeira et al.] Caldeira, A. B., Grabowski, B., Haug, V., Kahle, M.-E., Laidlaw, A., Maciel, C. D., Sanchez, M., and Sung, S. Y. Ibm power system s822.

[22] Caldeira, A. B., Grabowski, B., Haug, V., Kahle, M.-E., Maciel, C. D., and Sanchez, M. (2014). Ibm power systems s814 and s824 technical overview and introduction. *IBM Redbook REDP-5097-00*.

[23] Chasapis, D., Schulz, M., Casas, M., Ayguadé, E., Valero, M., Moretó, M., and Labarta, J. (2016). Runtime-Guided Mitigation of Manufacturing Variability in Power-Constrained Multi-Socket NUMA Nodes.

[24] Chiba, T., Yoshimura, T., Horie, M., and Horii, H. (2018). Towards selecting best combination of sql-on-hadoop systems and jvms. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 245–252. IEEE.

[25] Coles, H., Qin, Y., and Price, P. (2014). Comparing Server Energy Use and Efficiency Using Small Sample Sizes. Technical Report LBNL-6831E, 1163229.

[26] Coley, G. (2012). Beaglebone rev a6 system reference manual. *Obtenido*, 4(23):2012.

[27] Colmant, M., Rouvoy, R., Kurpicz, M., Sobe, A., Felber, P., and Seinturier, L. (2018a). The next 700 cpu power models. *Journal of Systems and Software*, 144:382–396.

[28] Colmant, M., Rouvoy, R., Kurpicz, M., Sobe, A., Felber, P., and Seinturier, L. (2018b). The next 700 CPU power models. *Journal of Systems and Software*, 144.

[29] Couto, M., Pereira, R., Ribeiro, F., Rua, R., and Saraiva, J. (2017). Towards a green ranking for programming languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages*, pages 1–8.

[30] Desrochers, S., Paradis, C., and Weaver, V. M. (2016). A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, page 455–470, New York, NY, USA. Association for Computing Machinery.

[31] Echtler, F. and Häußler, M. (2018). Open source, open science, and the replication crisis in hci. association for computing machinery, new york, ny, usa, 1–8.

[32] Eddie Antonio Santos, Carson McLean, Christophr Solinas, and Abram Hindle (2017). How does docker affect energy consumption? Evaluating workloads in and out of Docker containers. *The journal of systems & Software*.

[33] Efron, B. (2000). The bootstrap and modern statistics. *Journal of the American Statistical Association*, 95(452).

[34] El Mehdi Diouri, M., Gluck, O., Lefevre, L., and Mignot, J.-C. (2013). Your cluster is not power homogeneous: Take care when designing green schedulers!

[35] Fahad, M., Shahid, A., Manumachu, R. R., and Lastovetsky, A. (2019). A comparative study of methods for measurement of energy of computing. *Energies*, 12(11):2204.

[36] Fernandes, B., Pinto, G., and Castor, F. (2017). Assisting Non-Specialist Developers to Build Energy-Efficient Software. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 158–160.

[37] Fieni, G., Rouvoy, R., and Seinturier, L. (2020). Smartwatts: Self-calibrating software-defined power meter for containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 479–488. IEEE.

[38] Fieni, G., Rouvoy, R., and Seiturier, L. (2021). Selfwatts: On-the-fly selection of performance events to optimize software-defined power meters. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 324–333. IEEE.

[39] Gajewski, M. and Zabierowski, W. (2019). Analysis and Comparison of the Spring Framework and Play Framework Performance, Used to Create Web Applications in Java. In *2019 IEEE XVth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 170–173.

[40] Ge, R., Feng, X., Song, S., Chang, H.-C., Li, D., and Cameron, K. W. (2009). Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671.

[41] Goodman, S. N., Fanelli, D., and Ioannidis, J. P. (2016). What does research reproducibility mean? *Science translational medicine*, 8(341):341ps12–341ps12.

[42] Guimarães, M., Saraiva, J., and Belo, O. (2016). Some heuristic approaches for reducing energy consumption on database systems. *DBKDA 2016*, page 59.

[43] Hackenberg, D., Ilsche, T., Schöne, R., Molka, D., Schmidt, M., and Nagel, W. E. (2013). Power measurement techniques on standard compute nodes: A quantitative comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204. IEEE.

[44] Hackenberg, D., Ilsche, T., Schuchart, J., Schöne, R., Nagel, W. E., Simon, M., and Georgiou, Y. (2014). Hdeem: high definition energy efficiency monitoring. In *2014 Energy Efficient Supercomputing Workshop*, pages 1–10. IEEE.

[45] Hackenberg, D., Schöne, R., Ilsche, T., Molka, D., Schuchart, J., and Geyer, R. (2015). An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE international parallel and distributed processing symposium workshop*, pages 896–904. IEEE.

[46] Hammouda, A., Siegel, A. R., and Siegel, S. F. (2015). Noise-Tolerant Explicit Stencil Computations for Nonuniform Process Execution Rates. *ACM Transactions on Parallel Computing*, 2(1):1–33.

[47] Hankins, T. L. (1986). A debate over experiment: Leviathan and the air-pump. hobbes, boyle, and the experimental life. steven shapin and simon schaffer. including a translation of hobbes's dialogus physicus de natura aeris by simon schaffer. princeton university press, princeton, nj, 1985. xiv, 442 pp., illus. 60. *Science*, 232(4753):1040–1042.

[48] Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., and Hindle, A. (2016). Energy Profiles of Java Collections Classes. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 225–236.

[49] Heinrich, F., Carpen-Amarie, A., Degomme, A., Hunold, S., Legrand, A., Orgerie, A.-C., and Quinson, M. (2017). Predicting the Performance and the Power Consumption of MPI Applications With SimGrid.

[50] Hirst, J. M., Miller, J. R., Kaplan, B. A., and Reed, D. D. (2013). Watts up? pro ac power meter for automated energy recording.

[51] Ilsche, T., Hackenberg, D., Graul, S., Schone, R., and Schuchart, J. (2015). Power measurements for compute nodes: Improving sampling rates, granularity and accuracy. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, Las Vegas, NV, USA. IEEE.

[52] Inadomi, Y., Ueda, M., Kondo, M., Miyoshi, I., Patki, T., Inoue, K., Aoyagi, M., Rountree, B., Schulz, M., Lowenthal, D., Wada, Y., and Fukazawa, K. (2015). Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing.

[53] Islam, S., Noureddine, A., and Bashroush, R. (2016). Measuring energy footprint of software features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4. IEEE.

[54] Jagroep, E., Procaccianti, G., van der Werf, J. M., Brinkkemper, S., Blom, L., and van Vliet, R. (2017). Energy efficiency on the product roadmap: an empirical study across releases of a software product. *Journal of Software: Evolution and process*, 29(2):e1852.

[55] Jagroep, E., van der Werf, J. M. E., Jansen, S., Ferreira, M., and Visser, J. (2015). Profiling energy profilers. In *Proceedings of the 30th annual ACM symposium on applied computing*, pages 2198–2203.

[56] Joakim v Kisroski, Hansfreid Block, John Beckett, Cloyce Spradling, Klaus-Dieter Lange, and Samuel Kounev (2016). Variations in CPU Power Consumption.

[57] Johnson, S. C. (1967). Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254.

[58] Kalibera, T., Mole, M., Jones, R. E., and Vitek, J. (2012). A black-box approach to understanding concurrency in dacapo. In Leavens, G. T. and Dwyer, M. B., editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 335–354. ACM.

[59] Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K., and Ou, Z. (2018). Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2).

[60] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution.

[61] Koomey, J. et al. (2011). Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, 9(2011):161.

[62] Kothari, N. and Bhattacharya, A. (2009). Joulemeter: Virtual machine power measurement and management. *MSR Tech Report*.

[63] Kumar, M., Li, Y., and Shi, W. (2017). Energy consumption in Java: An early experience. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, Orlando, FL. IEEE.

[64] Kurpicz, M., Orgerie, A.-C., and Sobe, A. (2016). How much does a vm cost? energy-proportional accounting in vm-based environments. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 651–658. IEEE.

[65] Laros, J. H., Pokorny, P., and DeBonis, D. (2013). Powerinsight-a commodity power measurement capability. In *2013 International Green Computing Conference Proceedings*, pages 1–6. IEEE.

[66] LeBeane, M., Ryoo, J. H., Panda, R., and John, L. K. (2015). Watt watcher: fine-grained power estimation for emerging workloads. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 106–113. IEEE.

[67] Lengauer, P., Bitto, V., Mössenböck, H., and Weninger, M. (2017). A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008. In Binder, W., Cortellessa, V., Koziolek, A., Smirni, E., and Poess, M., editors, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 3–14. ACM.

[68] Libič, P., Bulej, L., Horky, V., and Tůma, P. (2014). On the limits of modeling generational garbage collector performance. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, page 15–26, New York, NY, USA. Association for Computing Machinery.

[69] Lilja, D. J. (2005). *Measuring computer performance: a practitioner's guide*. Cambridge university press.

[70] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: Reading kernel memory from user space.

[71] Liu, K., Pinto, G., and Liu, Y. D. (2015). Data-Oriented Characterization of Application-Level Energy Optimization. In Egyed, A. and Schaefer, I., editors, *Fundamental Approaches to Software Engineering*, volume 9033, pages 316–331. Springer Berlin Heidelberg, Berlin, Heidelberg.

[72] Longo, M., Rodriguez, A., Mateos, C., and Zunino, A. (2019). Reducing energy usage in resource-intensive Java-based scientific applications via micro-benchmark based code refactorings. *Computer Science and Information Systems*, 16(2):541–564.

[73] Lovicott, D. (2009). Thermal design of the dell™ poweredge™ t610™, r610™, and r710™ servers. *Round Rock. Texas*.

[74] Manotas, I., Sahin, C., Clause, J., Pollock, L., and Winbladh, K. (2013). Investigating the impacts of web servers on web application energy usage. In *2013 2nd International Workshop on Green and Sustainable Software (GREENS)*, pages 16–23.

[75] Marathe, A., Zhang, Y., Blanks, G., Kumbhare, N., Abdulla, G., and Rountree, B. (2017). An empirical survey of performance and energy efficiency variation on Intel processors.

[76] Margery, D., Morel, E., Nussbaum, L., Richard, O., and Rohr, C. (2014). Resources Description, Selection, Reservation and Verification on a Large-scale Testbed.

[77] McCreary, H.-Y., Broyles, M. A., Floyd, M. S., Geissler, A. J., Hartman, S. P., Rawson, F. L., Rosedahl, T. J., Rubio, J. C., and Ware, M. S. (2007). Energyscale for ibm power6 microprocessor-based systems. *IBM Journal of Research and Development*, 51(6):775–786.

[78] Mishra, S. K., Puthal, D., Sahoo, B., Jayaraman, P. P., Jun, S., Zomaya, A. Y., and Ranjan, R. (2018). Energy-efficient vm-placement in cloud data center. *Sustainable computing: informatics and systems*, 20:48–55.

[79] Morabito, R. (2015). Power Consumption of Virtualization Technologies: An Empirical Investigation. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pages 522–527.

[80] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276.

[81] Noureddine, A., Bourdon, A., Rouvoy, R., and Seinturier, L. (2012). A preliminary study of the impact of software engineering on GreenIT. In *2012 First International Workshop on Green and Sustainable Software (GREENS)*, pages 21–27.

[82] Noureddine, A., Islam, S., and Bashroush, R. (2016). Jolinar: analysing the energy footprint of software applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 445–448.

[83] Noureddine, A., Rouvoy, R., and Seinturier, L. (2015). Monitoring energy hotspots in software. *Automated Software Engineering*, 22(3):291–332.

[84] Oi, H. (2011). Power-performance analysis of jvm implementations. In *ICIMU 2011: Proceedings of the 5th international Conference on Information Technology & Multimedia*, pages 1–7. IEEE.

[Oliveira et al.] Oliveira, W., Oliveira, R., Castor, F., Fernandes, B., and Pinto, G. Recommending Energy-Efficient Java Collections. page 11. Chapter 2:
Collections- Introducing the main study field
- categorizing them with the characteristics of each category .

[86] Omair, A. et al. (2014). Sample size estimation and sampling techniques for selecting a representative sample. *Journal of Health specialties*, 2(4):142.

[87] O'Neil, E. J. (2008). Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356.

[88] Ournani, Z., Belgaid, M. C., Rouvoy, R., Rust, P., Penhoat, J., and Seinturier, L. (2020). Taming energy consumption variations in systems benchmarking. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 36–47, New York, NY, USA. Association for Computing Machinery.

[89] Patil, P. and Parmigiani, G. (2018). Training replicable predictors in multiple studies. *Proceedings of the National Academy of Sciences*, 115(11):2578–2583.

[90] Peng, R. D. (2011). Reproducible research in computational science. *Science*, 334(6060):1226–1227.

[91] Pereira, R., Carcao, T., Couto, M., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017a). Helping Programmers Improve the Energy Efficiency of Source Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 238–240, Buenos Aires, Argentina. IEEE. ## main idea
This paper presents a technique to spot the energy leaks inside a program using a spectrum technique called SPELL (Spectrum-based Energy Leak Localization)
How it works
We have a matrix (n*m) where n is the number of the tests and m is the number of components (like classes, methods, packages, etc.) after this we calculate the oracle (a vector that says which component is responsible for what)
Contributions
This paper helps developers to spot the red areas in their code and optimize the energy consumption. As an example it helped to reduce the energy consumption of a java application 50% faster and with 18% more efficiency.

[92] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017b). Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA. ACM. this paper discuss the energy consumption through memory usage however we aim to see the relation between CPU usage and energy consumption rather than memory usage.

[93] Philippot, O., Anglade, A., and Leboucq, T. (2014). Characterization of the energy consumption of websites: Impact of website implementation on resource consumption. In *ICT for Sustainability 2014 (ICT4S-14)*, pages 171–178. Atlantis Press.

[94] Pinto, G. and Castor, F. (2017). Energy Efficiency: A New Concern for Application Software Developers. *Commun. ACM*, 60(12):68–75. things to extend :
use micro benchmarks , and say that each one is used to measure a specific thing (memory intensive , cpu intensive , io intesive ..etc).

[95] Pinto, G., Liu, K., Castor, F., and Liu, Y. D. (2016). A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 20–31, Raleigh, NC, USA. IEEE.

[96] Prokopec, A., Rosa, A., Leopoldseder, D., Duboscq, G., Truma, P., Studener, M., Bulej, L., Zheng, Y., Villazon, A., Simon, D., Wurthinger, T., and Binder, W. (2019a). Renaissance: Benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 31–47, New York, NY, USA. Association for Computing Machinery.

[97] Prokopec, A., Rosà, A., Leopoldseder, D., Duboscq, G., Tuma, P., Studener, M., Bulej, L., Zheng, Y., Villazón, A., Simon, D., Würthinger, T., and Binder, W. (2019b).

Renaissance: benchmarking suite for parallel applications on the JVM. In *PLDI*, pages 31–47. ACM.

[98] Redondo, J. M. and Ortin, F. (2015). A Comprehensive Evaluation of Common Python Implementations. *IEEE Software*, 32(4):76–84. benchmarks link http://www.reflection.uniovi.es/python.

[99] Ribic, H. and Liu, Y. D. (2016). Aequitas: Coordinated energy management across parallel applications. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–12.

[100] Shapiro, S. S., Wilk, M. B., and Chen, H. J. (1968). A comparative study of various tests for normality. *Journal of the American statistical association*, 63(324):1343–1372.

[101] Shiv, K., Chow, K., Wang, Y., and Petrochenko, D. (2009). Specjvm2008 performance characterization. In *SPEC Benchmark Workshop*, pages 17–35. Springer.

[102] Simakov, N. A., Innus, M. D., Jones, M. D., White, J. P., Gallo, S. M., DeLeon, R. L., and FOPTurlani, T. R. (2018). Effect of meltdown and spectre patches on the performance of HPC applications. *CoRR*, abs/1801.04329.

[103] Stodden, V., Seiler, J., and Ma, Z. (2018). An empirical analysis of journal policy effectiveness for computational reproducibility. *Proceedings of the National Academy of Sciences*, 115(11):2584–2589.

[104] Tschanz, J., Kao, J., Narendra, S., Nair, R., Antoniadis, D., Chandrakasan, A., and De, V. (2002). Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *IEEE Journal of Solid-State Circuits*, 37(11).

[105] van der Kouwe, E., Andriesse, D., Bos, H., Giuffrida, C., and Heiser, G. (2018). Benchmarking Crimes: An Emerging Threat in Systems Security. *arXiv:1801.02381 [cs]*.

[106] van der Kouwe, E., Andriesse, D., Bos, H., Giuffrida, C., and Heiser, G. (2018). Benchmarking Crimes: An Emerging Threat in Systems Security. *CoRR*, abs/1801.02381.

[107] van Kessel, J., Taal, A., and Grosso, P. (2016). Power efficiency of hypervisor-based virtualization versus container-based virtualization. *University of Amsterdam*.

[108] Varsamopoulos, G., Banerjee, A., and Gupta, S. K. S. (2009). Energy Efficiency of Thermal-Aware Job Scheduling Algorithms under Various Cooling Models. In *Contemporary Computing*, volume 40. Springer.

[109] Vasan, A., Sivasubramaniam, A., Shimpi, V., Sivabalan, T., and Subbiah, R. (2010). Worth their watts? - an empirical study of datacenter servers.

[110] Vasques, T. L., Moura, P., and de Almeida, A. (2019). A review on energy efficiency and demand response with focus on small and medium data centers. *Energy Efficiency*, 12(5):1399–1428.

[111] Wang, B., Chen, C., He, L., Gao, B., Ren, J., Fu, Z., Fu, S., Hu, Y., and Li, C.-T. (2018a). Modelling and developing conflict-aware scheduling on large-scale data centres. *Future Generation Computer Systems*, 86:995–1007.

[112] Wang, Y., Nörtershäuser, D., Le Masson, S., and Menaud, J.-M. (2018b). Potential effects on server power metering and modeling. *Wireless Networks*.

[Wang et al.] Wang, Y., Nörtershäuser, D., Masson, S. L., and Menaud, J.-M. Experimental Characterization of Variation in Power Consumption for Processors of Different generations. page 10.

[114] Zar, J. H. (2005). Spearman rank correlation. *Encyclopedia of biostatistics*, 7.

[115] Zimmerman, D. W. (1987). Comparative power of student t test and mann-whitney u test for unequal sample sizes and variances. *The Journal of Experimental Education*, 55(3):171–174.