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 October 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
October 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

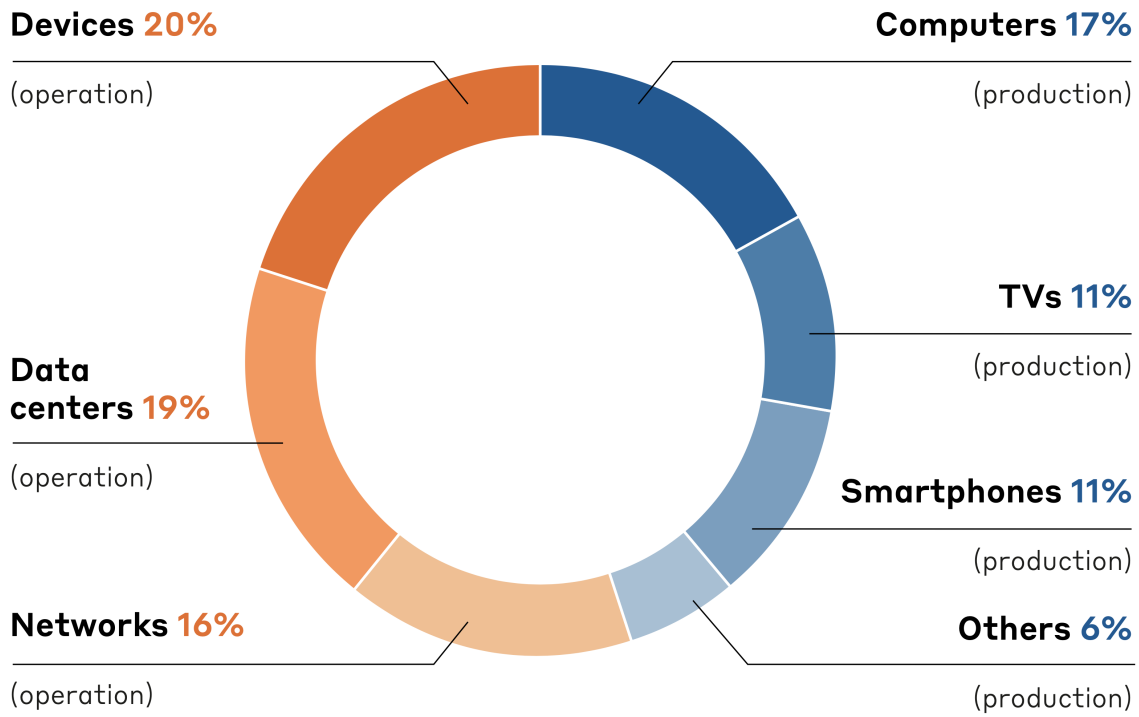


Figure 1.1: Final energy consumption of digital technologies by item in 2017

[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. **??**: Where we discuss the work done on the energy consumption and optimization in software engineering
2. **??**: It will present a set of guidelines and tools to help practitioners measure the energy consumption of their algorithms.
3. : it will discuss the behaviour of python and the possibile ways to tune it in ordrer to reduce the energy consumpion
4. : will present a study on java programming language and the impact of the JVM choice on the energy consumpion
5.
6. : we will present the impact of programming languages on the energy consumption of the algorithms specially when it comes to web services .
7. : 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 most 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 **??** 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, **?** 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 [**?**]. 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.

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

[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.

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 intended to assist practitioners in better comprehending and optimizing the energy usage of their applications. Thus, we will begin by examining 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 we can finally adapt them to energy consumption.

Section **??** will discuss the pitfalls and best practices associated with empirical research before applying them to our field of interest. After that Section **??** 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, representing a significant obstacle to achieving precise readings and higher accuracy. Then, in Section **??**, 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 also be applicable outside the research lab. 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 researchers face. Reproducing an experiment has been one of the significant 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 [**?**]. 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 [**?**]. 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 [**?**]. According to [**?**], *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 **??** 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. **?** presents a guideline on achieving 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. However, many attempts have been made to provide a set of benchmarks for a 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 minor, self-contained, and easy to implement in any language. The benchmarks are also made to represent most of the typical real-world workloads in an isolated manner. Dacapo [**?**] and renaissance [**?**] 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 [**?**], 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 system's validity.

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 **?** , 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 [**?**]. Another study that was published in the book "Measuring computer performance: a practitioner's guide" [**?**],**?** examined performance indicators and gave an in-depth treatment of benchmark program tactics. They clearly explained 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.

? 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 [**?**] 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 [**?**] 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 [**?**], 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, do not 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 simultaneously.

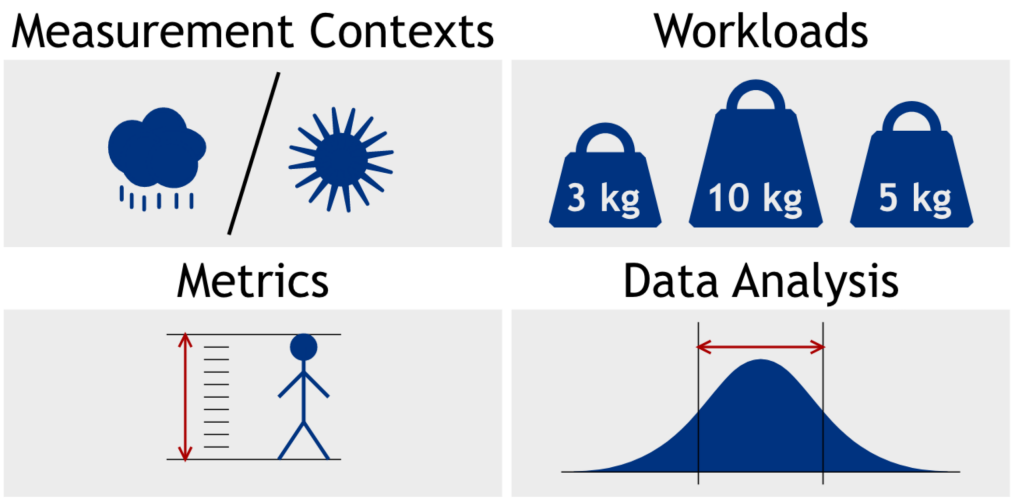


Figure 2.2: The proposed decomposition of an experiment [**?**]

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

1. *measurement contexts* indicates the software and hardware components that will alter or remain constant during the experiment.
2. *workloads* identify the benchmarks to use in the experiment, as well as their inputs;
3. *metrics* 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 **?** where they provided a tool to highlight the most energy consuming parts of the code [**?**]. The essential advantage of this approach is that it allows practitioners 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 **?** , there are four main criteria to evaluate an energy measurement tool [**?**]:

- *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 into our system,*
- Accuracy: to eliminate extra hazards and get a more representative measurement.

We believe that accuracy can be extracted from these criteria by combining the first two results. Therefore, we will focus more on the first three criteria later on. Below, we will discuss some of the well-known tools used in literature.

2.3.1 Hardware Tools

Nowadays, most high-performance computing systems (HPC) implement a tool to report the nodes' energy consumption for 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 energy consumption history. 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 shallow sampling frequency. Most of those tools are provided directly by the manufacturers. Such as *IBM EnergyScale technology [**? ? ?**] or* Dell poweredge [**?**], MEGware Clusstafe [**?**]. 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 of up to 1 Hz. It has an internal memory to store a wide variety of data, such as the maximum voltage and current, that can later be exported via USB port for personal usage or lined to 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 [**?**] Despite its high temporal sampling, WattsUp Pro lacks spatial granularity since it monitors the energy consumption of the whole system. To have finer granularity, we need to isolate the energy consumption of each component.

PowerMon and its upgraded version powerMon2 [**?**] are based on a micro-controller chip that can simultaneously monitor up to 6 channels (8 for powermon2). Therefore, we can 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 is smaller and can fit within 3.5 inches rack drive.

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

powerpack [**?**] 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 [**?**], 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 **?** and**?** The CPU is the part responsible for the most energy consumption in a data center[**? ?**]. 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*) [**? ?**] is a set of registers that Intel introduced in their CPU since Sandy bridge generation, and later it was followed by AMD since Family 17h Zen.

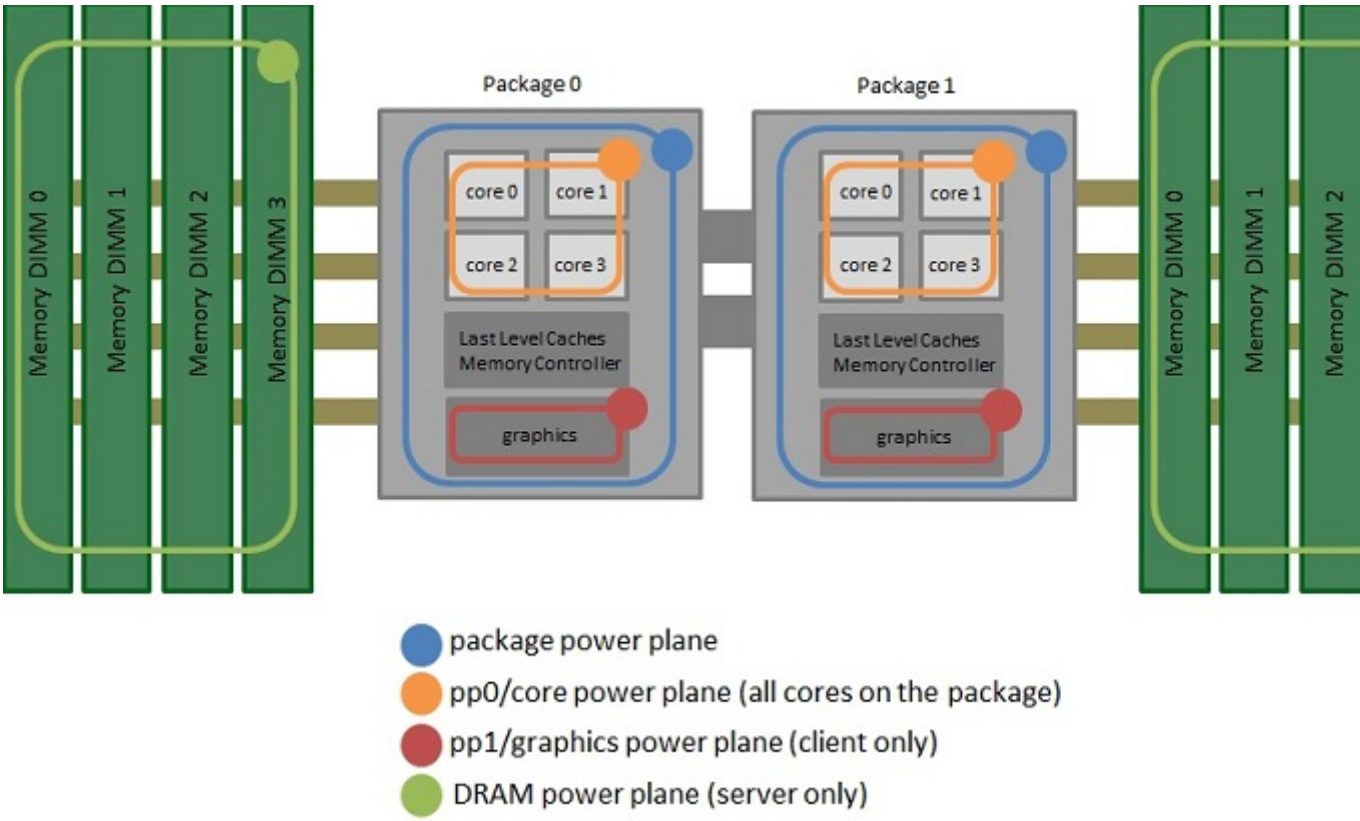


Figure 2.3: INTEL RAPL scopes

Figure **??** 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, while the integrated GPU is only available on desktop processors. The advantage of such an approach is the absence of intrusive measurement tools. Furthermore, they have a high temporal granularity with a sampling frequency of up to 1 KHz [**?**].

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

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 a power model's behavior and estimate energy usage. This model is then used to allocate the observed energy consumption among different execution entities, such as processes, control groups, threads, or code lines.

The first examples of software measurement tools are PowerAPI [**?**], SmartWatts formula [**?**] and SelfWatts [**?**]. 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 real-time energy usage data precision 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 functional control group. PowerAPI is only compatible with Linux on a bare-metal physical computer.
WattWatcher [**?**] 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 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 [**? ?**] 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 provides power-limiting features for virtual machines. Previous Joulemeter tests [**?**] 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 [**? ?**]. 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 [**?** **?**]. The tool does not need a calibration phase and relies on pre-established power models based on hardware metrics like TDP, disk I/O rate, and CPU frequency. These settings must be identified and supplied by the user for his machine. Jolinar can only measure one application's energy consumption at a time. At the end of the execution, the tool provides the process's CPU, DRAM, and disk energy usage. Jalen [**?**] 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 insufficient to reproduce the exact energy measurements between identical machines and even within the same machine. This difference—also called *energy variation* (EV)—has seriously threatened the accuracy of experimental evaluations.

Figure **??** 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 [**?**], on 4 nodes of an homogeneous cluster (the cluster Dahu described in Table **??**) at 50 % workload. We can observe a consedirable 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 [**?** **?**] who reported that the leading cause of the variation of the power consumption between different machines is due to the **CMOS** manufacturing process of transistors in a chip. [**?**] described this variation as a set of parameters, such as CPU Frequency and the thermal effect.

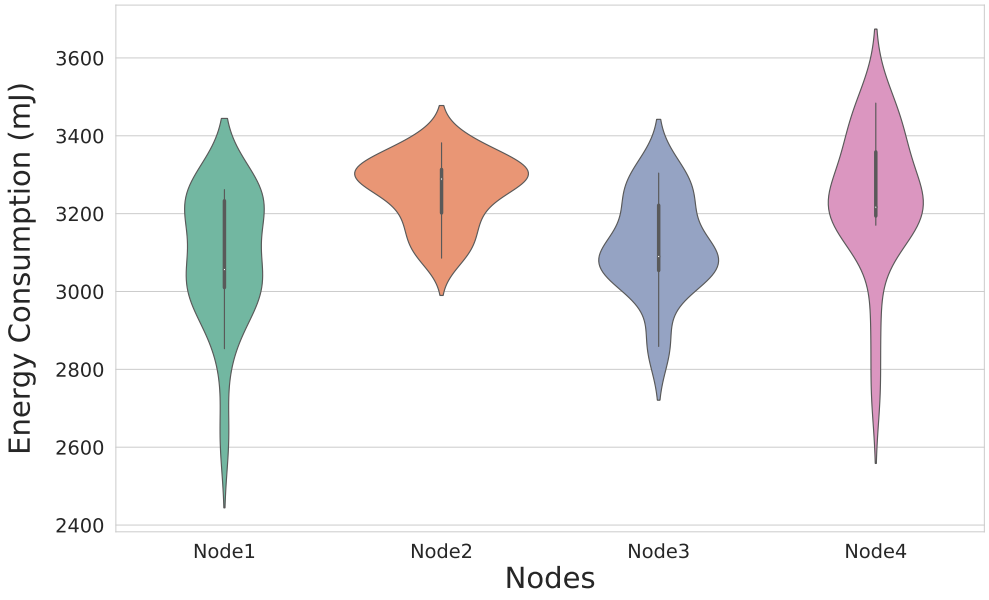


Figure 2.4: CPU energy variation for the benchmark CG

2.4.1 Studying Hardware Factors

This variation has often been related to the manufacturing process [**?**]. However, it 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 [**?**] that identical processors could exhibit significant energy consumption variation with no close correlation with the processor temperature and performance. On the other hand, the authors of [**?**] 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 when comparing energy consumption variations across homogeneous processors.

The ambient temperature was also discussed in many papers as an essential candidate factor for the energy variation of a processor. In [**?**], the authors claimed that energy consumption might 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 significant variations from one node to another.

In [**?**], 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 [**?**], 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. **?** [**?**] used three different power measurement tools: RAPL, Power Insight[11] and BGQ EMON. All 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, many papers proposed contributions to reduce and mitigate this variation. In [**?**], 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 [**?**] proposed parallel algorithms that tolerate the variability and the non-uniformity by decoupling per process communication over the available CPU. Acun *et al.* [**?**] 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, 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 [**?**], the researchers showed how a parallel system could deal with the energy variation by compensating for the uneven effects of power capping.

In [**?**], 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.

2.5 Energy Optimizations

,

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

We will now focus on 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 considerable 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 focus on the following two 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. The goal is to make the final product use less energy by choosing the best 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 running.

We start with the work of **?** where they did an energy consumption comparison analysis of the most popular programming languages. The paper recommends combining some of these languages to enhance code quality while considering 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 to improve energy economy, execution speed, and memory utilization. This paper's findings were based on the game benchmark, the most famous benchmark comparing several programming languages.

This paper defines 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.

? investigated the influence of programming language choice on the energy consumption of software during execution. In their paper [**?**], 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 exciting situations where slower languages use less energy than faster ones, even though fast languages usually use the least energy. Finally, they produced an energy efficiency rating of programming languages. The paper [**?**] compared the energy, performance, and database response time of web applications written in Java versus those written in Kotlin. They discovered 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 have studied the impact of website technology on energy consumption, [**?** **?**]. In their work [**?**],**?** 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.

As for the impact of the source code on the energy consumption, we can cite [**?** **?**] 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.

? 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 could result in a 300% increase in energy consumption [**?**].

Other studies [**?** **?**] looked into the energy use of Java primitive types, string operations, and the use of exceptions, loops, and arrays. **?** , 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.

? [**?**] presented SPELL, a tool that helps developers spot energy leaks in their source code. Using a statistical spectrum-based analysis and JRAPL [**?** **?**], 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. The goal is to optimize this energy consumption for already developed software without changing the source code. The goal is to set up and create an environment where software can run with the least amount of energy. This could involve process scheduling, system tunning, and so on.

We begin with Aequitas [**?**], a system that allows parallel applications to live on co-managed power domains (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 (for 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, **?** analyzed the total energy consumption of a VM in a data center while emphasizing the fixed cost versus the dynamic one. In this paper [**?**], 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, including the air conditioning, power distribution, and the dynamic cost related to the VM activities.

The energy usage of virtual machine allocation and task placement has also been investigated [**?**].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. In order to solve the problem of high heterogeneity of activities and resources, the jobs are categorized based on their resource requirements. 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 that The suggested technique saves energy by reducing the number of active PMs while minimizing the makespan and task rejection rate.

Besides the impact of virtual machine orchestration, some works have been targeting the runtime of specific programming languages. Using the TPC-DS benchmark, **?** 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 [**?**]. **?** also compared the performance of HOTSPOT and J9. They demonstrated in their research [**?**] that the workload affects the relative performance of a JVM. 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 [**?**] benchmarks.

As for python, **?** 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. At the same time, Python3 versions covered more tests due to compatibility and the fact that Python2 became obsolete.[**?**].

2.6 Conclusion

As we have seen in state of the art, many methods have been proposed to reduce the ICT's energy footprint, which can be applied in different parts of the program's lifecycle, from consumption to execution. Furthermore, the execution phase took the attention of many researchers because it is the part where the most energy is consumed. This thesis will focus

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

on that aspect as well. However, unlike most work done on the hardware aspect, we will target the software impact on this energy, starting from the choice of the programming language to how to tune some features of a framework 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 algorithm's complexity, 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. executes the code,
2. measure the program,
3. infers the guidelines.

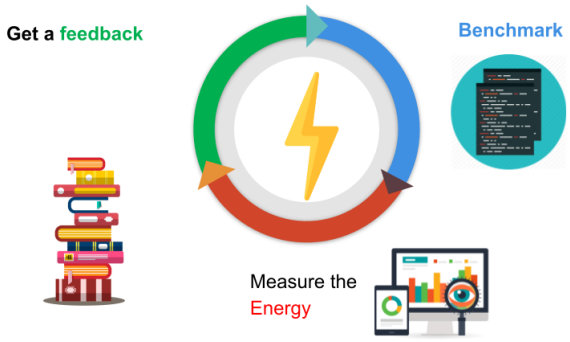


Figure 2.5: the spiral method of energy optimization

This work aims to present a set of guidelines to create a benchmarking system to measure different programs' energy consumption. After that, we will use this system to compare the energy consumption of different programming languages. We will extend the work of **?** to a closer distance to the production environment by comparing a set of use cases.

Chapter 3

Benchmarking Protocol to Measure Software Energy Consumption

This chapter covers ways to overcome empirical analysis challenges in energy consumption studies. First, we go through the three components of a successful benchmark when performing energy-related experiments. Section **??** focuses on the "reproducibility" challenge—to deliver reproducible experiments without interfering with the energy measurement—while Section **??** discusses the *accuracy* of software energy consumption.

3.1 Reproducibility within the context of energy

3.1.1 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 [**?**].

Second, one should not ignore tremendous progress in computer science, which has led to a rise in the number of obsolete results. Furthermore, when it comes to comparative research, the execution environment may impact 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 to ensure that the results can be *reproduced*, so one can test their hypothesis in different environments and 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 propose a protocol to ensure that the results are reproducible and extensible.

3.1.2 Virtual Machines

The first choice should be to use *Virtual Machines* (VM). This technology enables researchers to choose the most appropriate tools, software, and operating system for their needs without incurring the cost of changing the 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 replicated by instantiating a copy of the image.

This option, however, comes at a cost. Because of the hypervisor, the 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, this interaction may be required—-especially if the experiment is dependent on an external source such as energy monitors.

3.1.3 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 **??** 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 with 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 servers and VirtualBox are notable examples of this category, and most researchers' experiments use them. However, the applications are typically slower because of the two operating systems.

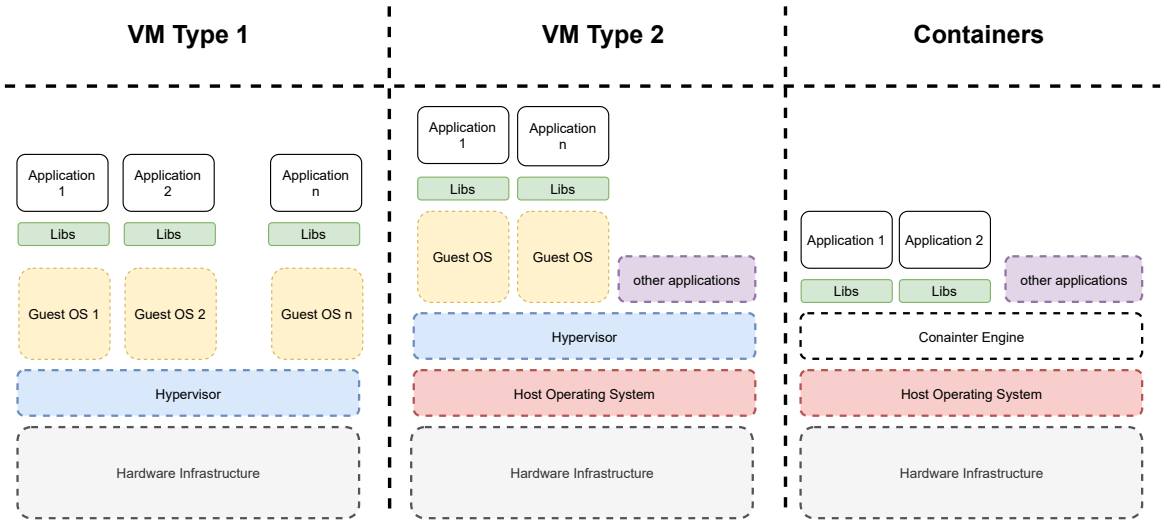


Figure 3.1: Different Methods of Virtualization

- 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* [**?**].

3.1.4 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 [**? ?**], we require a lightweight orchestrator to limit the overhead on the energy usage of our experiments
2. We need to communicate with the host OS because we are using hardware sensors to measure energy consumption.

3.1.5 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 [**?**] who examined how adding the Docker layer affected energy consumption, **?** conducted their experiment by running numerous benchmarks both with and without Docker. They contrasted the energy usage and execution time that resulted. The

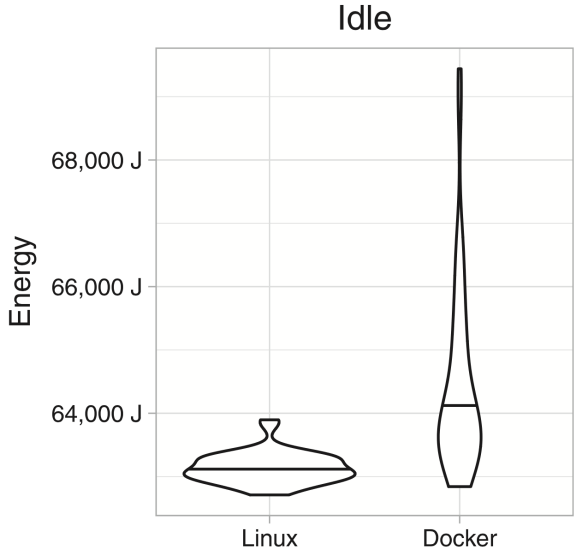


Figure 3.2: energy consumption of Idle system with and without docker [**?**]

first step was to observe the effects of the orchestrator and the Docker daemon while there was no work to be done. Then, they used three benchmarks in their experiments: WordPress, Redis, 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 **??**.

However, as seen in Figure **??**, Docker increased the execution duration of the benchmark by 50 seconds, which led to a significant rise in energy usage. According to the authors, the Docker daemon is primarily responsible for this overhead, not the fact that the application is running 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 compared to the benefits of Docker for isolation and reproducibility.

To summarize, Docker-based software consumes 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 addition of the supervisor has increased Docker's impact on energy, which will be distributed equitably across 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, ensuring that they can be repeated clearly and simply.

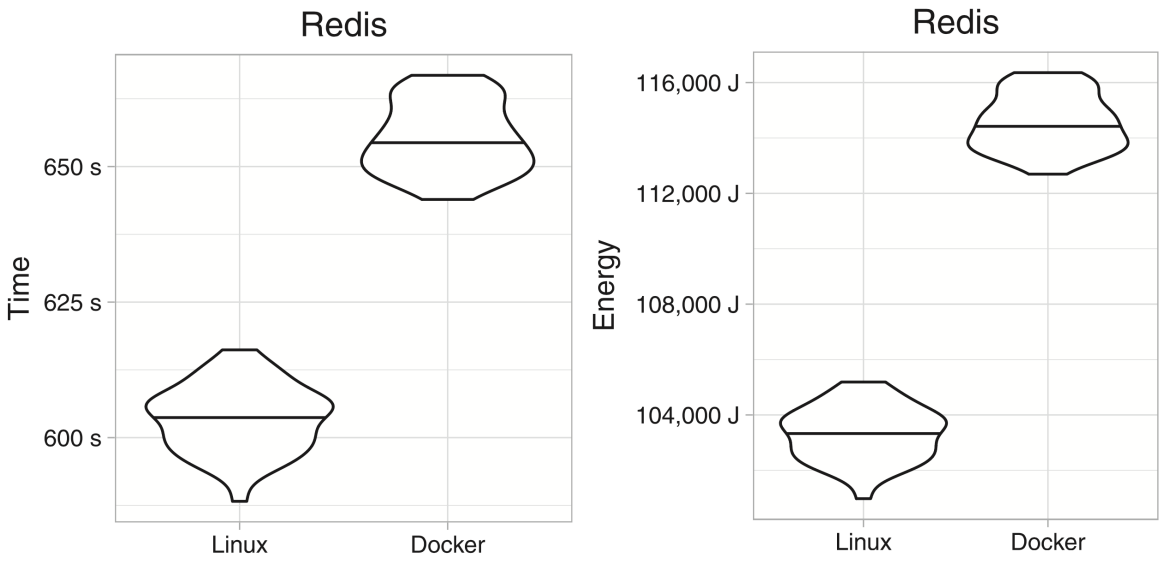


Figure 3.3: Execution time & energy consumption of Redis with and without Docker [**?**]

3.1.6 Extension

Definition of Extention

With the rapid evolution of the software industry, Even ensuring the reproducibility of the same research will not be enough. Each day, new software versions 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 expand 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 performance, cost, and security, among others.
- **benchmarks**: This axe aims to enlarge the benchmarks to include others to broaden the research scope.

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/

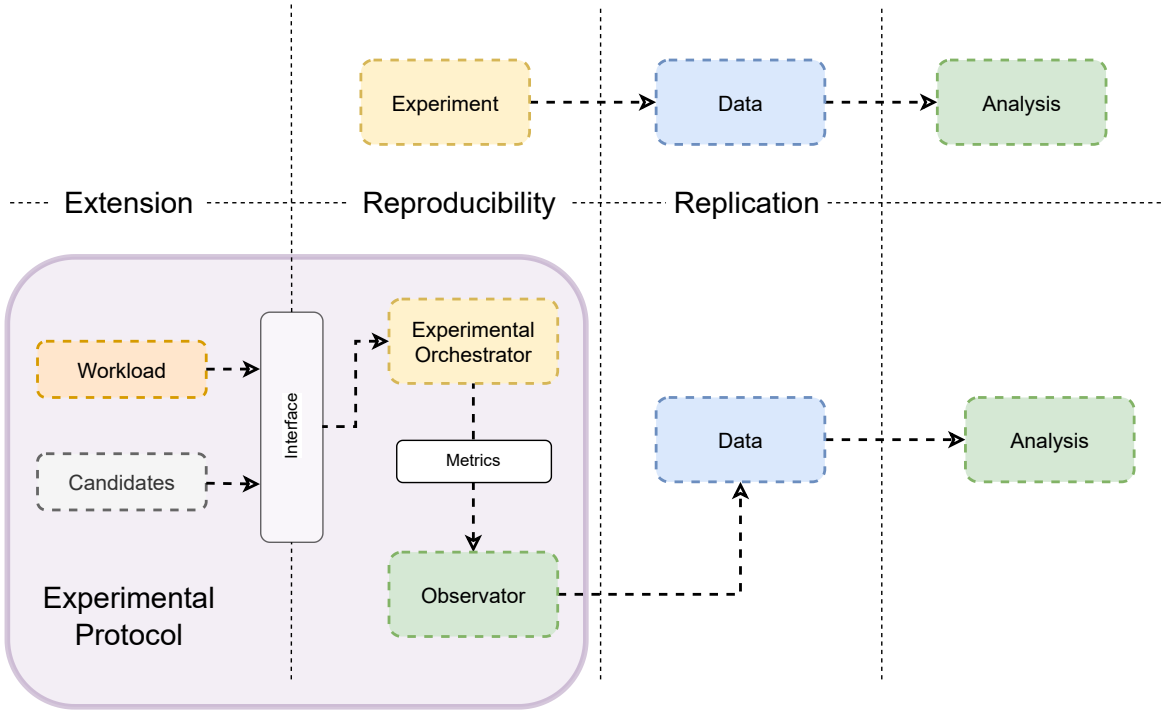


Figure 3.4: Benchmarking protocol

aspects of their guidelines that were mentioned previously in Section **??**, we suggest an abstract model describing an empirical experiment. Figure **??** shows the proposed model while comparing 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 better judge the results and reproduce the experiment while diminishing the impact of external factors.

orchestrator The core design of the experiment is responsible for running the experiment regardless 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 the workload should implement to be called by each candidate in the experiment. This interface is responsible for extending the experiment with new benchmarks.
- *Observer interface*: provides a set of metrics that the orchestrator collects 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 the candidate should implement to called by the orchestrator. This interface is responsible for extending the experiment with new solutions.

Data The row data collected by the experiment. In our case, it will be provided by the *observator*. One or more observators can be used to collect different metrics. This data aims to ensure the *replication*, allowing the reader to perform extra analysis without executing the experiment a second time.

Analysis The final part should provide the set of methods and functions used to do the empirical analysis to answer the research questions.

3.1.7 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 using VMs is unsuitable for performance or energy-related tests, while containers are a good alternative. Later, we discussed the benefits and drawbacks of using Docker for energy consumption experiments. We have demonstrated that it has a constant overhead that is self-mitigated when compared. Lastly, we have proposed an addition to the benchmarking framework that would allow the experiment to be extended along the three axes already mentioned.

3.2 Improving Accuracy by Taming Energy Variations

3.2.1 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.

3.2.2 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?

3.2.3 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) [**? ?**], a large, flexible testbed for experiment-driven research spread across France, to meet these needs. Grid5000 has several clusters comprising 4 to 124 identical machines with different configurations for each cluster. We

looked at four groups for our experiment, and our main criterion was the CPU configuration. Table **??** 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*. Our study considers hyper-threads as distinct **PUs**. Figure **??** shows the detailed topology of a node in the cluster Dahu as an example.

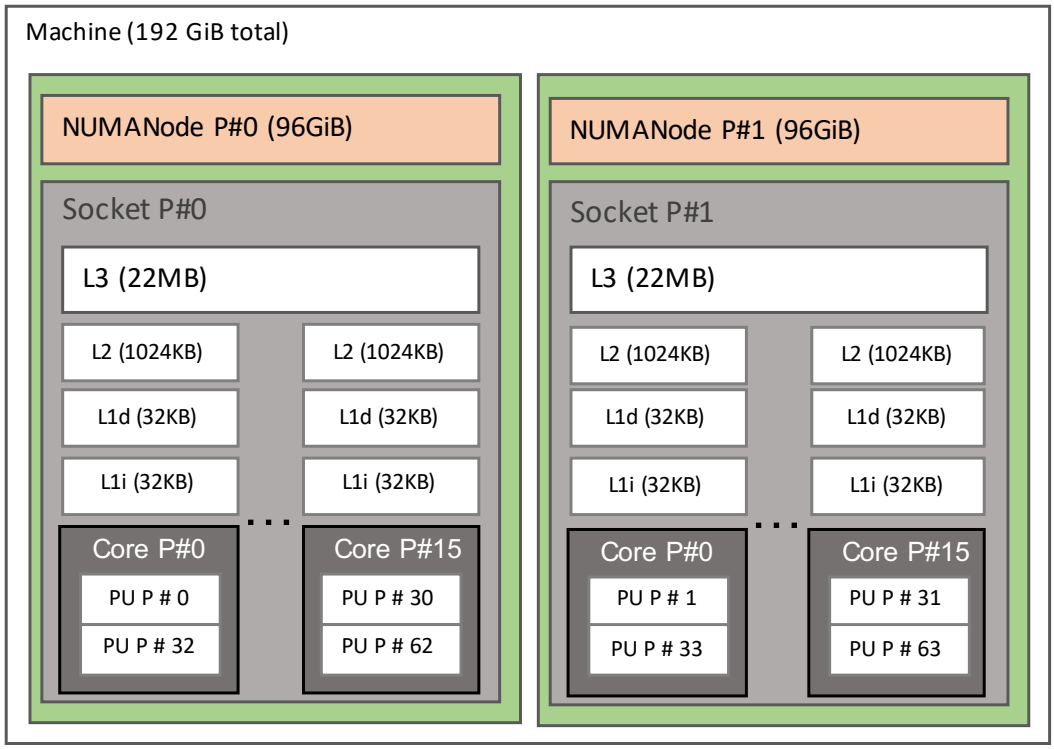


Figure 3.5: Topology of the nodes of the cluster Dahu

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 [**?**].

To meet these criteria, we looked at the "state of the art" and found the most common benchmarks used to test hardware performance. We then chose the ones that could be scaled up. Our candidate is `NAS Parallel Benchmark` (NPB v3.3.1) [**?**]: 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. We first need a tool to measure energy consumption to study this variation. For this we used POWERAPI [**?**], which is a power monitoring toolkit that is based on *Intel Running Average Power Limit* (RAPL) [**?**]. 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. Docker 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 sought 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. While running our experiments, we tried not to fall into the most common benchmarking "crimes" [**?**]. As we study the STD difference of measurements we observed from empirical experiments, we use the bootstrap method [**?**] to randomly build multiple subsets of data from the original dataset, and we draw the STD density of those sets, as illustrated in Figure **??**. 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 three different workloads in our experiments: single process, 50 %, and 100 %, to cover the low, medium, and high CPU usage when analyzing the effect of the studied parameters, respectively. These workloads reflect the ratio of used PU count to the total available PU.

3.2.4 Analysis

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

3.2.5 Docker & Accuracy

As the state of the art assesses the impact of Docker on 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 three nodes of the cluster Dahu to assess if Docker induces an additional variation. Figure **??** reports that this is not the case, as the energy consumption variation does not get noticeably affected by Docker while running the same compiled version of the benchmarks at 5 %, 50 % and 100 % workloads. While Docker increases the energy consumption due to the extra layer it implements [**?**], 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

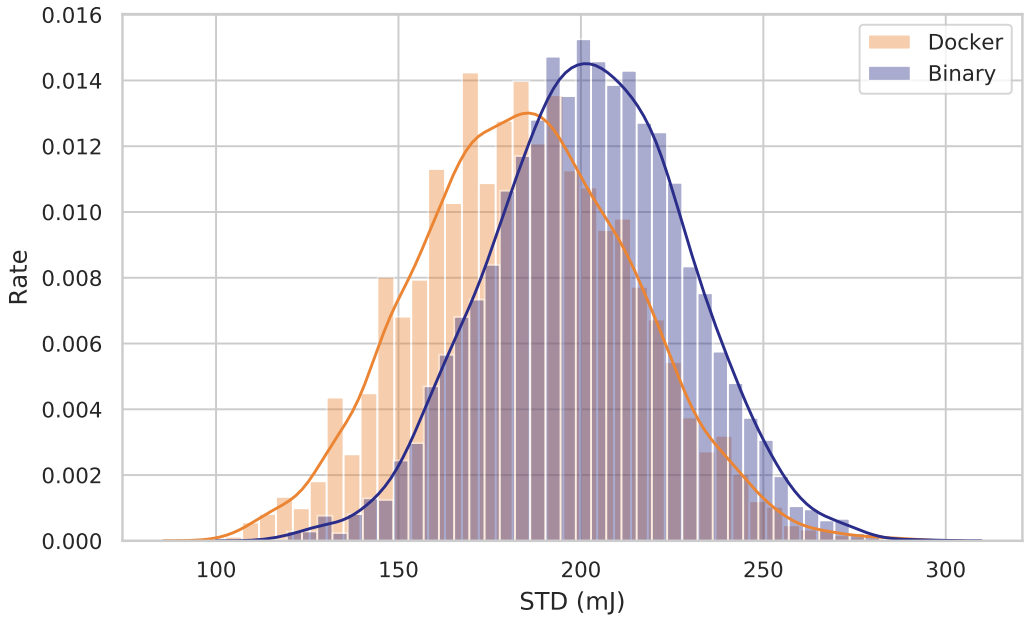


Figure 3.6: Comparing the variation of binary and Docker versions of aggregated LU, CG and EP benchmarks

RQ 1: Benchmarking Protocol

To achieve a robust and reproducible experiment, practitioners often repeat their tests multiple times 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 three 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, after each iteration, the "reboot" mode automatically reboots the machine. The difference between the normal and sleep modes highlights that the CPU needs rest before starting another iteration, especially for an intense workload. Putting the CPU into sleep for several seconds could give it time to reach a lower frequency state or/and reduce its temperature, which could impact the energy variation. On the other hand, the reboot mode 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 registries. However, the

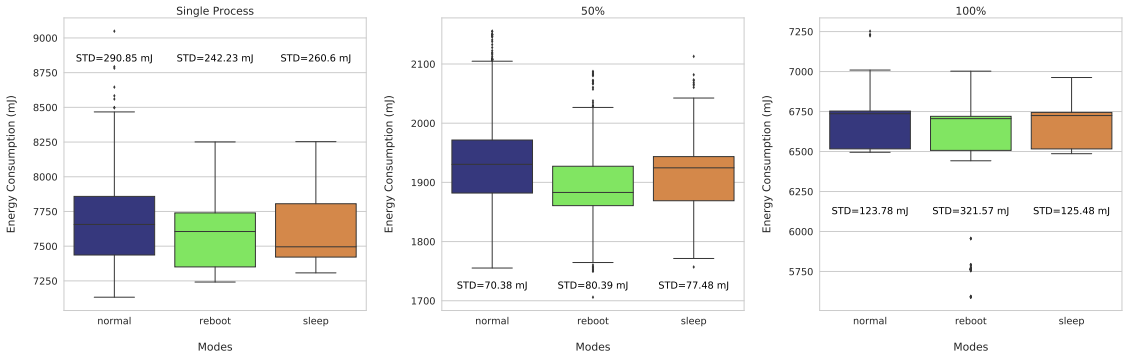


Figure 3.7: Energy variation with the normal, sleep and reboot modes

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. However, it deserves to be checked to investigate if it effectively enhances the overall energy variation or not.

Figure **??** reports on 300 aggregated executions of the benchmarks LU, CG and EP, on 4 machines of the cluster Dahu (cf. Table **??**) for different workloads. We note that the results have been executed with different datasets sizes (B, C and D for a single process, 50 % and 100 % respectively) to remedy the brief execution times at high workloads for small datasets. This justifies the scale differences in reported energy consumption between the three modes in Figure **??**. One can observe that picking one of these strategies does not strongly impact the energy variation for most workloads. All the strategies 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 a substantial amount of outliers. This goes against our expectations, even when setting a warm-up time after reboot to stabilize the OS.

In Figure **??**, 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 higher than the two other modes. This makes the reboot mode 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 in high workloads.

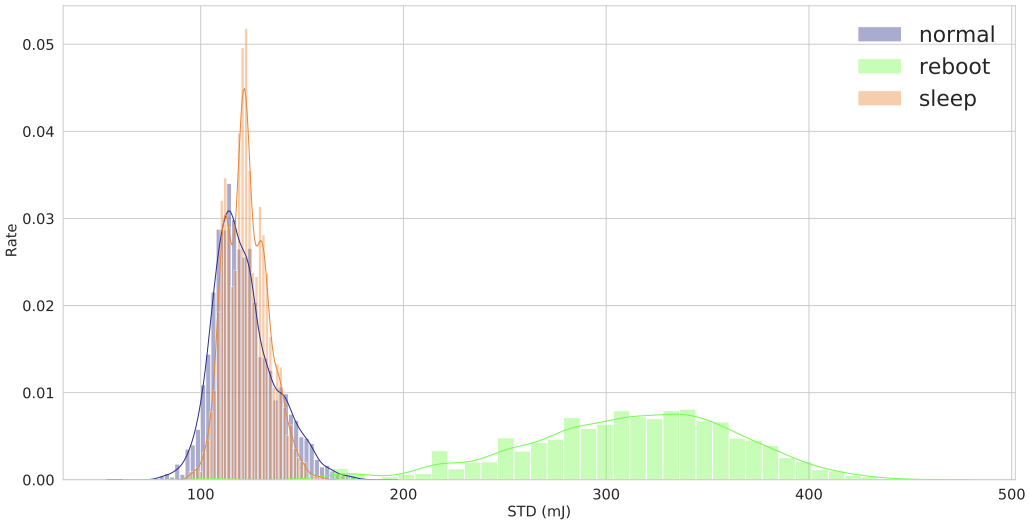


Figure 3.8: STD analysis of the normal, sleep and reboot modes

RQ 2: Processor Features

The C-states allow switching the CPU between more or less consuming states upon activities. Turning the C-states on or off have been the subject of many discussions [**?**] because of its dynamic frequency mechanism, but, to the best of our knowledge, there has been no entirely 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 **??** depicts the results of the experiments we executed on three nodes of the cluster Dahu. On each node, we ran the same 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 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 a vital 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

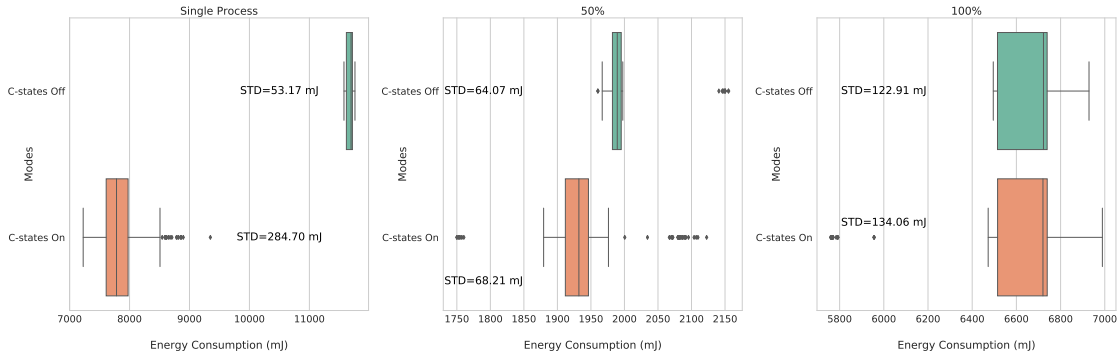


Figure 3.9: Energy variation when disabling the C-states

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 not to affect the total energy consumption nor its variation. 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 **??**, but reduces the variation by 5 times compared to the C-states on mode. This leads to two questions: Can a process pinning method reduce/increase the energy variation? Moreover, how does the energy consumption variation evolve at different PU usage levels?

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 its core usage (all the cores are being used). The third strategy (S3) also involves splitting the workload between the two sockets, but with the hyper-threads on each core used, i.e., half of the cores are used across the two CPUs.

Figure **??** 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

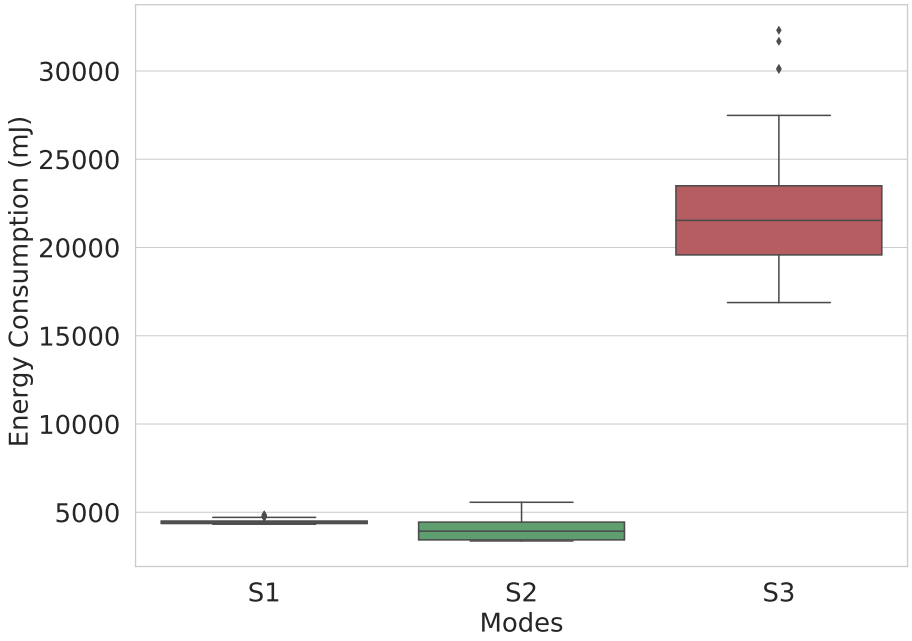


Figure 3.10: Energy variation considering the three cores pinning strategies at 50 % workload

the benchmarks `LU` and `EP`). For example, S2 is the least power-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 strategies, 32 processes are being scheduled on 16 physical cores using the hyper-threads PU, which will introduce more context switching, thus, more energy consumption.

Even though the first and third strategies are very similar (both use hyper-threads, but only on one CPU for the first and two CPUs for the third), the variation gap between them is significant, as the variation in the first strategy is 30 times lower ($STD_{S1} = 116mJ$,$STD_{S3} = 3,452mJ$). This shows that 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 reports low energy consumption because distributing the threads across all the cores completes the task faster than in the other cases. Hence, it consumes less energy. The S3 is a highly consuming strategy because both sockets are 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 for the worst results observed when using the third strategy is that recurrent context switching, which is added to the OS scheduling and can reschedule processes from one socket to another, invalidates cache usage because a process cannot benefit from the socket local L3 cache when it moves from one CPU to another. (cf. Figure **??**).

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 **??** 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. 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 understand better 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. 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, opposite the claims of [**?**].

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 **??** 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 disabling the CPU C-states does not decrease the variation for all the workloads; as we can

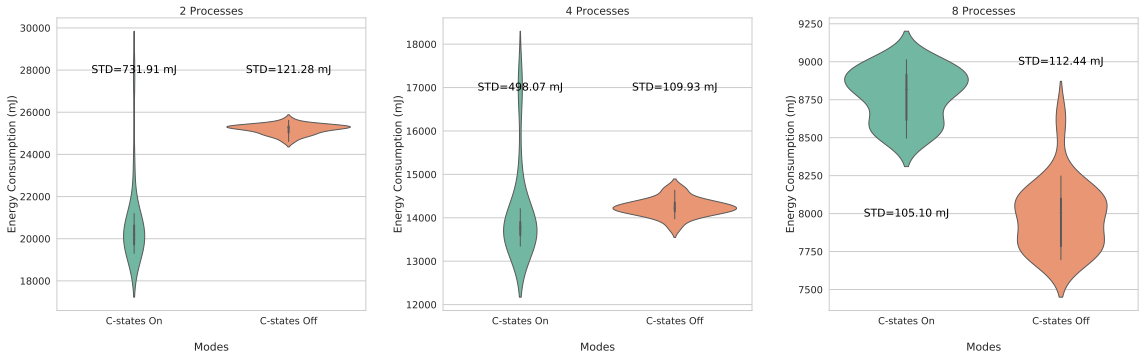


Figure 3.11: C-states effect on the energy variation, regarding the application processes count

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 four processes and almost equal to the C-states on mode when running eight processes. In fact, running more processes implies using 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 four 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 energy consumption while reducing the variation. However, 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 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 consumption of two algorithms or two software system versions is an example of a 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, thus boosting performance

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

under specific constraints. When demanding tasks run, the operating system decides to use the processor's highest performance state.

Disabling or enabling the Turbo Boost impacts the CPU frequency behavior directly, as enabling it allows the CPU to reach higher frequencies to execute some tasks for a short period. However, its usage does not significantly impact the energy variation. Acun *et al.* [**?**] 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 showed that disabling the Turbo Boost does not exhibit any considerable positive or negative effect on the energy variation. Table **??** compares the STD when enabling/disabling the Turbo Boost, where the columns combine workload and benchmark. We only had minor measurement differences when switching on and off the Turbo Boost. We favored or against using 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. When a CPU is used at its maximum capacity, the cores would heat up very fast and 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 for the energy variation. The variation difference is barely noticeable when disabling the Turbo Boost, which cannot be considered due to the OS activity and the measurement error margin. We cannot affirm that the Turbo Boost does not have an impact on all the CPUs, as we only tested on two recent Xeon CPUs (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 significant overhead and therefore a potential threat to the validity. The purpose of 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 this 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 **??**, 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 frequency states. However, it does not mean that this variation is 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 its idle behavior when running a single process.

Inspecting the OS idle energy variation is insufficient to relate the energy variation to the active job. The OS can behave differently regarding 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 [**?**] of the energy consumption and variation between global RAPL and POWERAPI, as illustrated in Figure **??**. 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 entirely 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 the Meltdown security

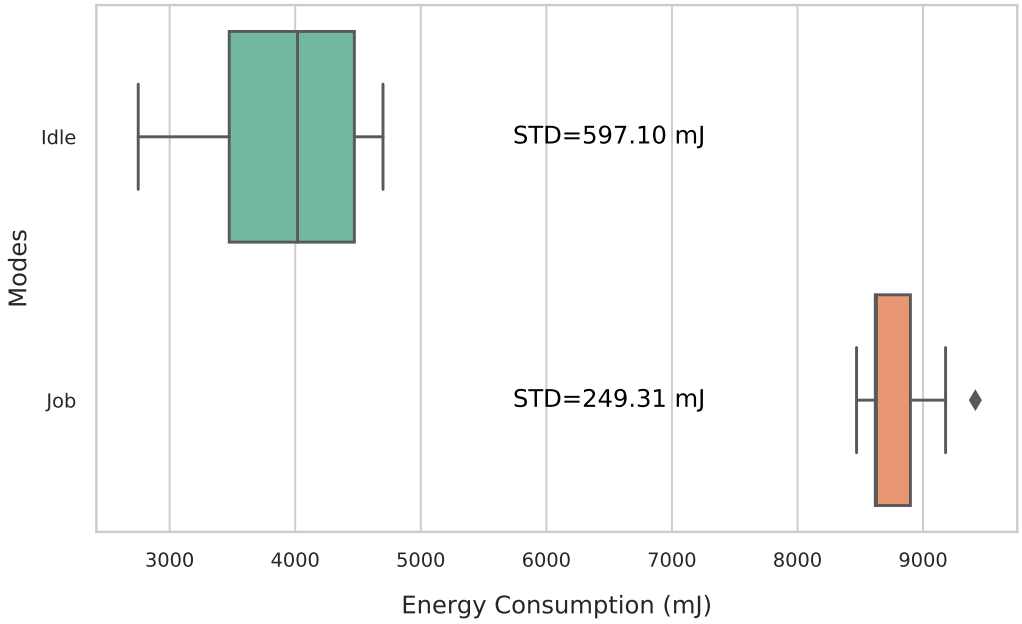


Figure 3.12: OS consumption between idle and when running a single process job

patch — known to be causing some performance degradation [**?** **?**]— affects the energy variation?

OS Tuning An OS is a pack of running processes and services that might or not be required for its execution. Even using a minimal version of Debian Linux, we could list many OS running services and processes 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. We only kept the strict minimum required for the experiment's execution. Table **??** 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 one node to another. Moreover, 10 % of the variation is not a representative difference due to many factors that can affect it as the CPU temperature or the measurement errors.

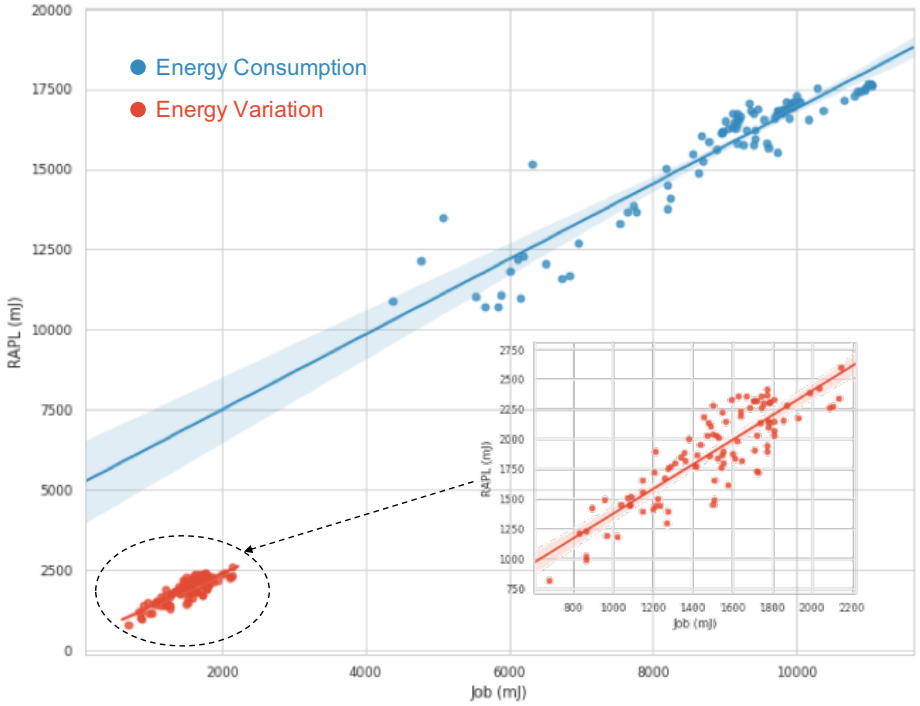


Figure 3.13: The correlation between the RAPL and the job consumption and variation

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 other processes' data that is supposed to be private [**?** **?**]. 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 have been applied to prevent/reduce the criticality of these vulnerabilities. On the Linux kernel, the patch has been automatically applied since 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 [**?**] 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 **??** 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. 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 [**?** **?**] might only be responsible for a slight 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 new CPUs have enhancements to the chip density, the maximum Frequency, or some optimization features like the C-states or the Turbo Boost. This dynamic evolution caused different generations of CPUs can handle a task differently. This experiment aims 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 [**?** **?**], which makes measurements stability even worse. In this experiment, we, therefore, compare four different generations of CPU to evaluate the energy variation for each CPU and its correlation with the generation. Table **??** indicates the characteristics of each tested CPU.

Table **??** 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 chip's energy behaviors. On the other hand, our experiments showed the lowest energy variation when using the Ecotype CPU, these CPUs 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 **??** is an illustration of the aggregated STD density of more than 5,000-random value sets taken from all the conducted experiments. This shows that the cluster Paranoia

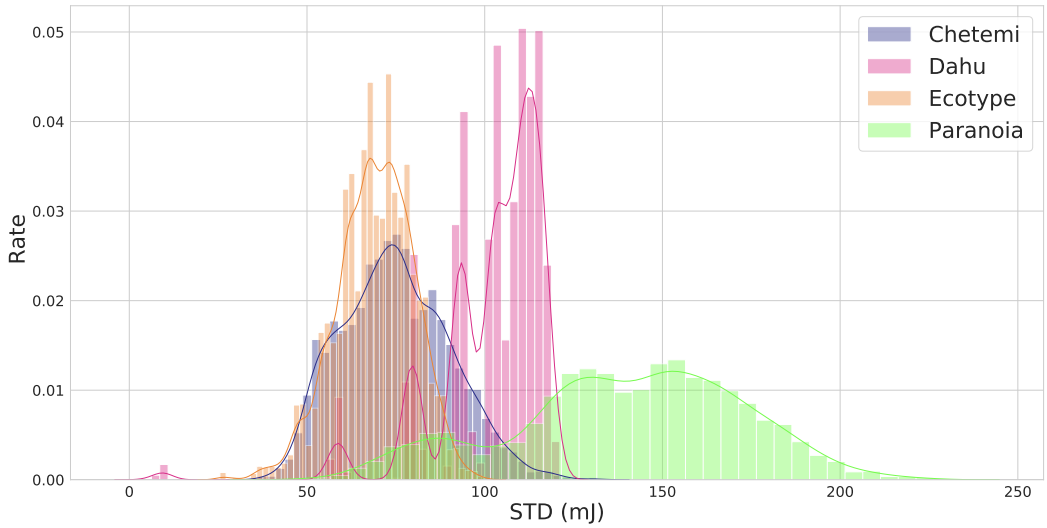


Figure 3.14: Energy consumption STD density of the 4 clusters

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 get fewer variations.

3.2.6 Experimental Guidelines

To summarize our experiments, we provide some practical guidelines in Table **??**, based on the multiple experiments and analyses 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. Therefore, it intends to help practitioners tame the energy variation on the selected CPU and conduct experiments with the tiniest variations.

Table **??** gives a proper understanding of known factors, like the C-states and their variation reduction at low workloads. However, it also lists some new factors we identified along the analysis we conducted in Section, such as the results related to the OS or the reboot mode. Some guidelines are more useful/efficient for specific workloads, as shown in our experiments. Thus, qualifying the workload before conducting the experiments can help choose the proper guidelines. Other studied factors, such as Turbo Boost or Speculative execution, are not mentioned in the guidelines due to the small effect observed in our study.

To validate the accuracy of our guidelines among a varied set of benchmarks on the one hand and their effect on the variation between identical machines, on the other hand, we ran

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 the same machine. Identical machines can exhibit many differences regarding their energy behavior	Any	Up to 1.3×

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 have 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 effect of the guidelines on the nodes where the variation is susceptible to be higher.

Figure **??** and **??** 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 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 ($Mean_{LU-normal-Dahu2} =$

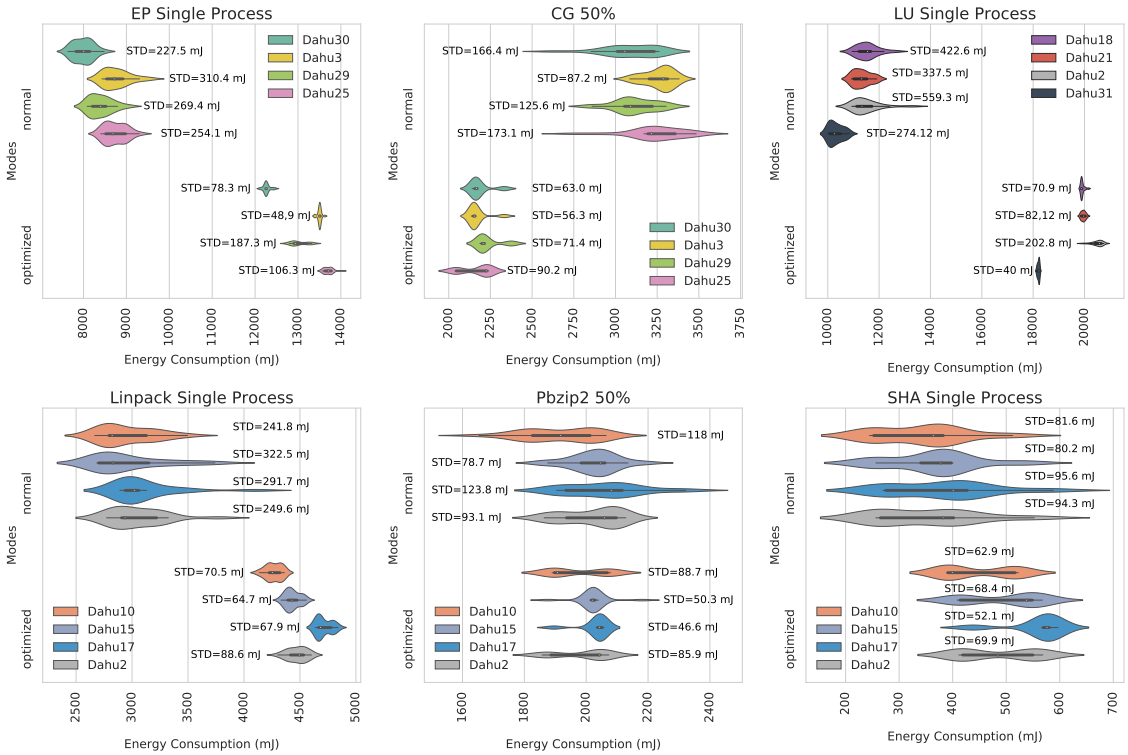


Figure 3.15: Energy variation comparison with/without applying our guidelines

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 fewer energy thanks to an appropriate core pinning method.

Figures **??** and **??** 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 [**?**]. However, taming the intra-node variation is a good strategy for identifying more relevant mediums and medians and then accurately comparing the nodes' variation. Even though using the same node is always better, it avoids the extra inter-nodes variation and thus improves the stability of measurements.

3.2.7 Threats to Validity

Several 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

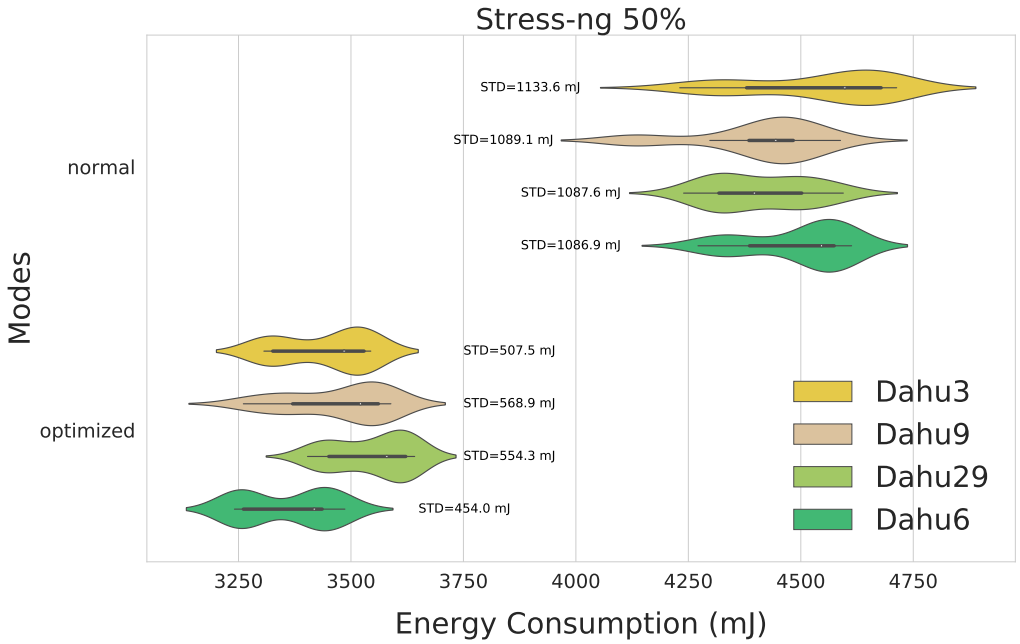


Figure 3.16: Energy variation comparison with/without applying our guidelines for STRESS-NG

the most accurate tools for modern CPUs but still adds a significant overhead if we adopt a sampling at high frequency. The other fine-grained measurement tool we used is POWERAPI. It allows measuring 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 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 material's micro-heterogeneity are among the most significant issues, as we cannot track or justify some energy variations between identical CPUs 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.

3.2.8 Summary

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 through controlled experiments. In this study, We investigated some previously discussed aspects of 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.

3.3 Conclusion

As seen in Section **??** of 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 reproducibility techniques. We have observed that there are two primary methods for encapsulating experimental systems, the first is to utilize virtual machines, and the second 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 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) to keep up with the rapid pace of software development. Furthermore, we have provided a model that enables comparative experiment *extension*. The foundation of this model is separating 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 energy-based experiments. We started by analyzing the impact of the chosen encapsulation method from the previous section on this challenge to determine 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` [**?**], on a variety of machines with diverse hardware combinations and operating system setups and tunings. We have shown that optimizing the operating system can significantly affect 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 results in the research environment cannot 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.

This chapter aims to establish a reproducible and accurate protocol for conducting energy-related experiments. From now on, we shall use this protocol in our studies to reduce software energy usage.

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

Chapter 4

Impact of Energy-saving Strategies in the Python Ecosystem

4.1 Introduction

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

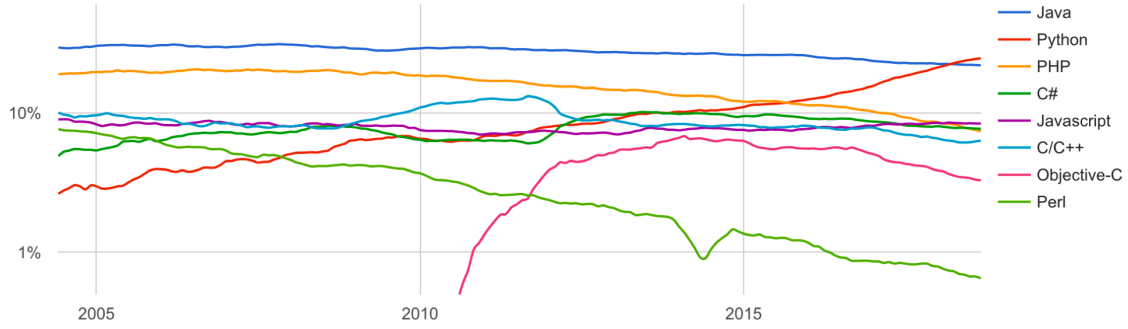


Figure 4.1: PYPL popularity of programming languages [**?**].

In particular, **?** [**?**] in 2012, and then Pereira *et al.* [**?**] in 2017, conducted empirical power measurements on this topic: 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 [**?**] (cf. **??**).

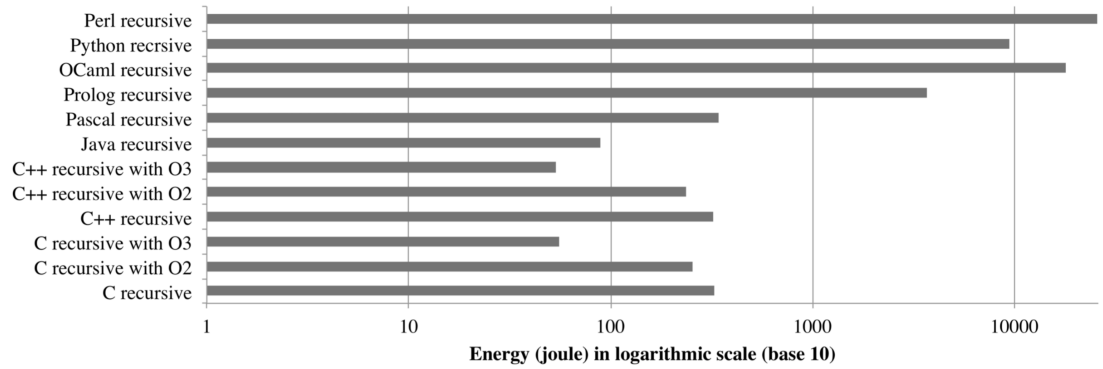


Figure 4.2: Energy consumption of a recursive implementation of Tower of Hanoi program in different languages [**?**].

In this chapter, we aim to reduce the energy consumption of Python code. We first start by presenting the motivation behind our work in Section **??**. Then, we share some insights on the impact of programmer choices. To do this, we report on a series of experiments to examine the energy consumption of Python code across various use cases. Section **??** will then end by investigating several features of Python structure and their impact on energy consumption. After that, in Section **??**, we investigate other non-intrusive approaches to optimizing the energy consumption of applications by comparing multiple Python runtime implementations, which include 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.

4.2 Motivation

4.2.1 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 consumption 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 mainly for energy considerations. Rather, we believe

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

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.

4.2.2 Python Gluttony

According to [**?**] and [**?**], Python tends to be one more energy hungry programming language. As one can notice in **??**, Python consumes 30 times more than C or C++. The benchmark was implemented with the Tower of Hanoi[2] of 30 disks.

As shown in **??**, 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]

Table 4.1: 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

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.

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

4.2.3 Python Use Cases

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. **??** illustrates a study published by the JetBrain company on Python developers.[4] 57% of the respondents reported that they use Python for data science, and 51% said they use it for web development. Around 40% are using it for system administration.[5]

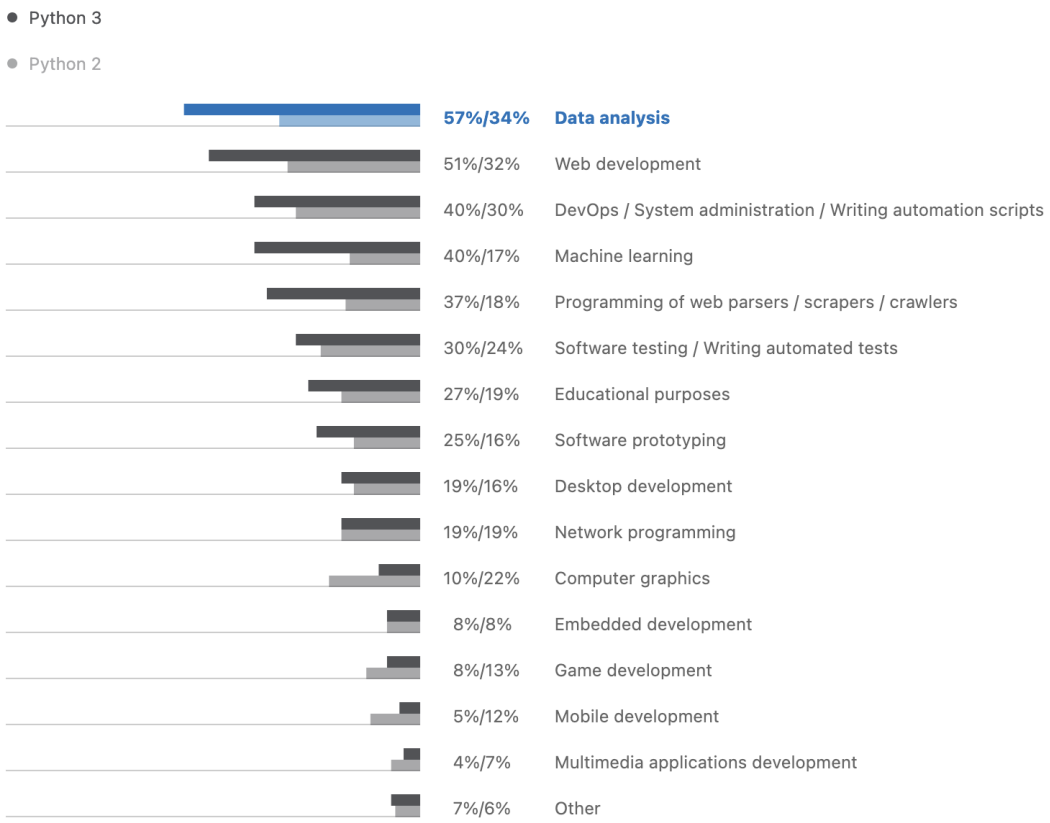


Figure 4.3: Use cases of Python (source: JetBrain).

[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%.

4.3 Optimizing the Energy Consumption of Python Applications

In this section, we study 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 **?** did in [**?**]. Then, we measure the energy consumption of several Python implementations to propose a non-intrusive technique to improve energy efficiency. Overall, this section will address the following research questions:

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: machine learning (cf. **??**) and web applications (cf. **??**). Then, we dive deeper into the energy consumption of Python core structures before concluding with the impact of parallelism on energy consumption.

4.3.1 Python for 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 significantly impact climate change, it is vital to investigate mitigation techniques.

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: Each experiment is done within a Docker container using Jupyter lab to ensure reproducibility. These tests are run atop a minimal version of Debian-10 to increase the tests' accuracy by eliminating unnecessary processes.

Input Workload

Models: Several models were developed. However, only two were used in the final trials because they achieved 94% accuracy with an acceptable training time. David Page's cifar10-fast[6] and Woonhyuk Baek's torch skeleton[7] are reported here.

Datasets: The CIFAR-10 dataset was the major source of data for the studies. It is made up of $60,000$ `32x32` color images grouped into 10 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 $60,000$ `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.

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. The key performance metrics are, therefore:

- *accuracy*: in %
- *execution time*: in seconds for both the duration of each epoch and the total duration to achieve a certain accuracy
- *total energy consumption*: in joules, including the CPU and the GPU.

Results & Findings

As the model's accuracy increases, so do the energy required for the next accuracy increment. Figure **??** 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.

[6]https://github.com/davidcpage/cifar10-fast
[7]https://github.com/wbaek/torchskeleton

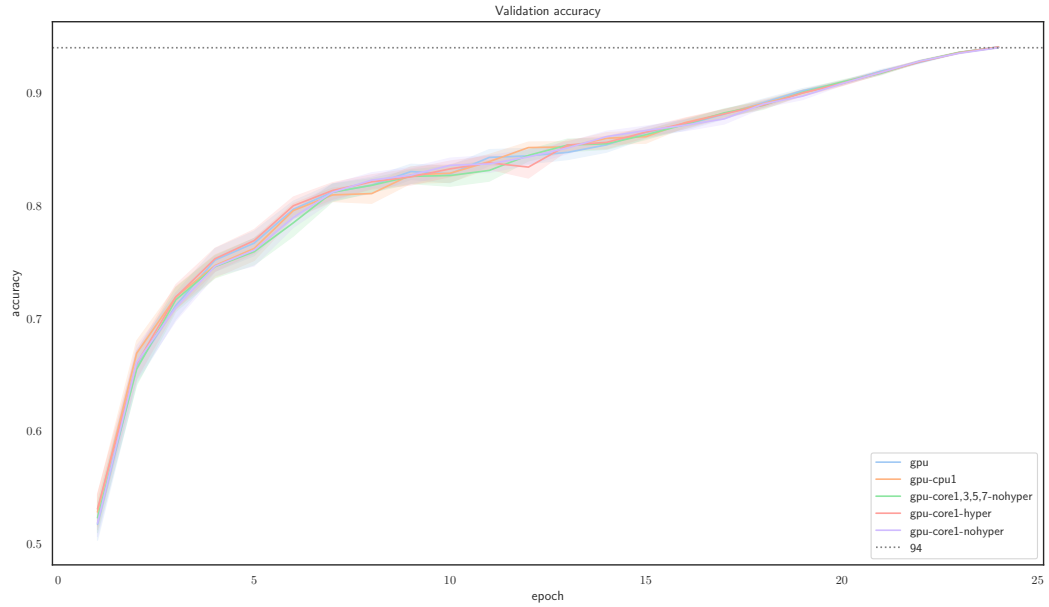


Figure 4.4: Accuracy along epochs.

?? shows the model's accuracy based on the number of epochs. This figure shows that there is a non-significant impact on the choice of the strategy on the accuracy; it is only a matter of the number of epochs. However, the increase in accuracy is not linear, as can be observed. It took only 10 epochs to reach an accuracy of 84%, but it required more than twice the number of epochs to add an extra 6% accuracy. This highlights the price one should pay to increase the model's accuracy.

?? confirm this observation. As one can see, the training of the model up to 90% accuracy requires three times the energy required for training to 80% accuracy. Moreover, one can notice that this price is paid mainly by the CPU and GPU, while the memory is not that impacted.

Interestingly there were no significant differences between different strategies—around 1.3%—as shown in Figure **??**. However, this gap may increase as the number of epochs increases.

On the other hand, as the number of epochs increases, the average power consumption decreases until it reaches a steady state after 10 epochs. This could be connected to the caching strategies adopted by the CPU and GPU. However, this evolution is still insignificant compared to the baseline value. This is shown in Figure **??**.

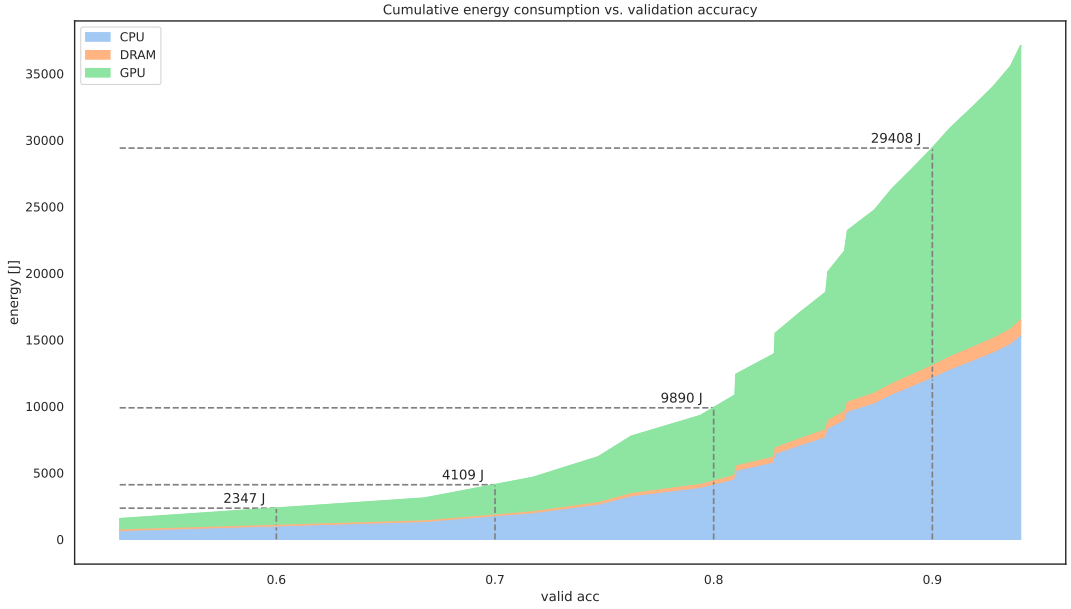


Figure 4.5: Cumulative energy consumption vs. accuracy.

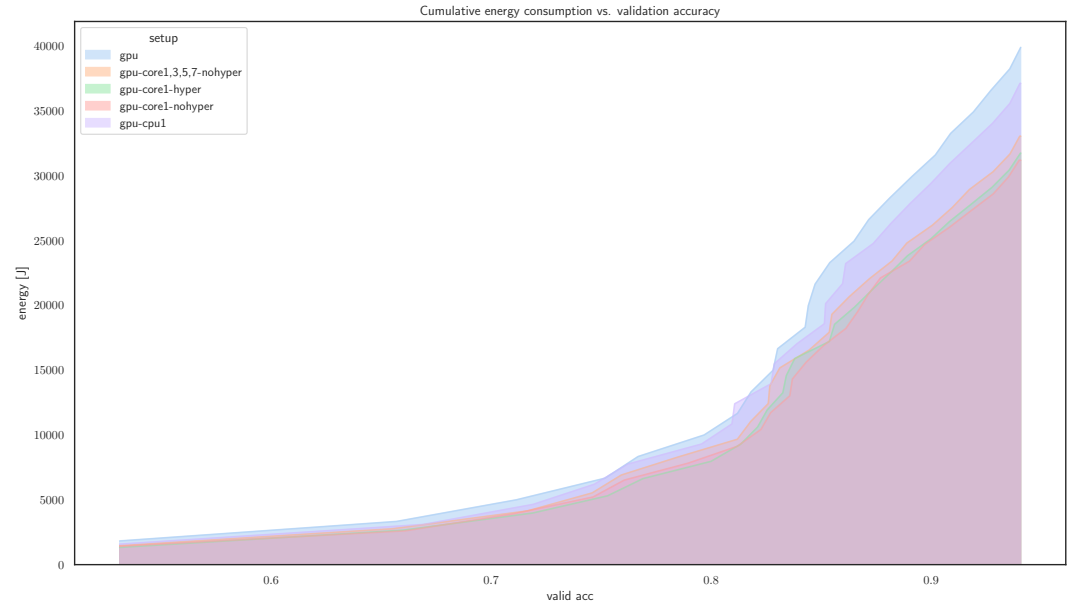


Figure 4.6: Average energy for training the model based on the strategy.

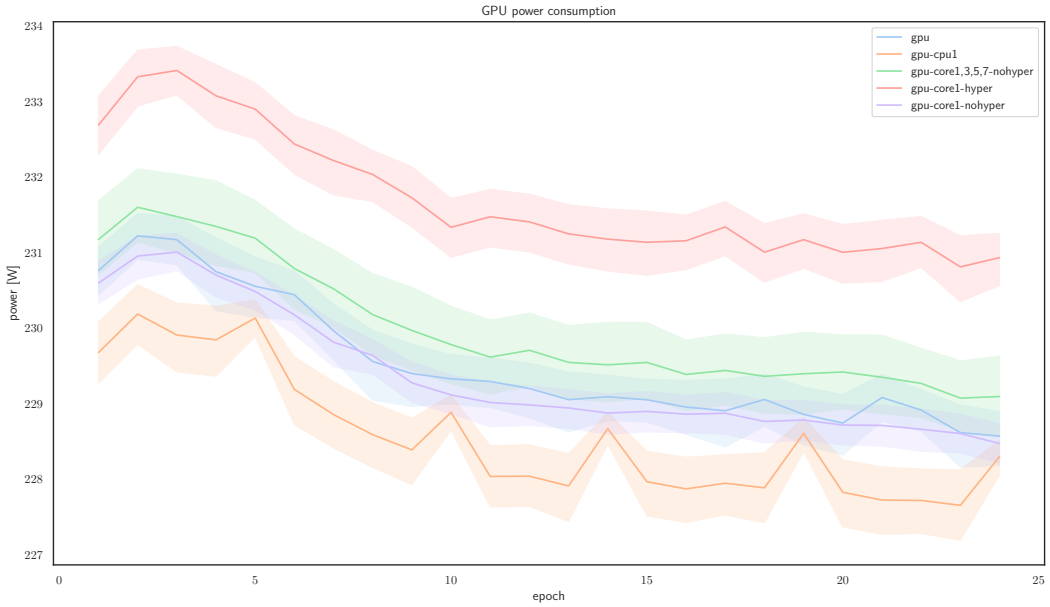


Figure 4.7: Evolution of average GPU power along epochs.

As one can see in Figures **??** and **??**, the average power consumption and the average duration of an epoch do not have a significant variation, as most of them last around 3.7 seconds. However, these two values have an intermediate correlation if we exclude the strategy based on one core and its hyper-thread. The last strategy exhibits more power- even when its duration is not optimal- because the context switches between the core and its hyper-thread.

Synthesis

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. Is a 10% increase in accuracy worth it if we have to spend three times the energy? While using the GPU, the CPU has no significant impact on the energy consumption of the training. Therefore, one can limit the number of CPU cores used during the execution without sacrificing the execution time and accuracy.

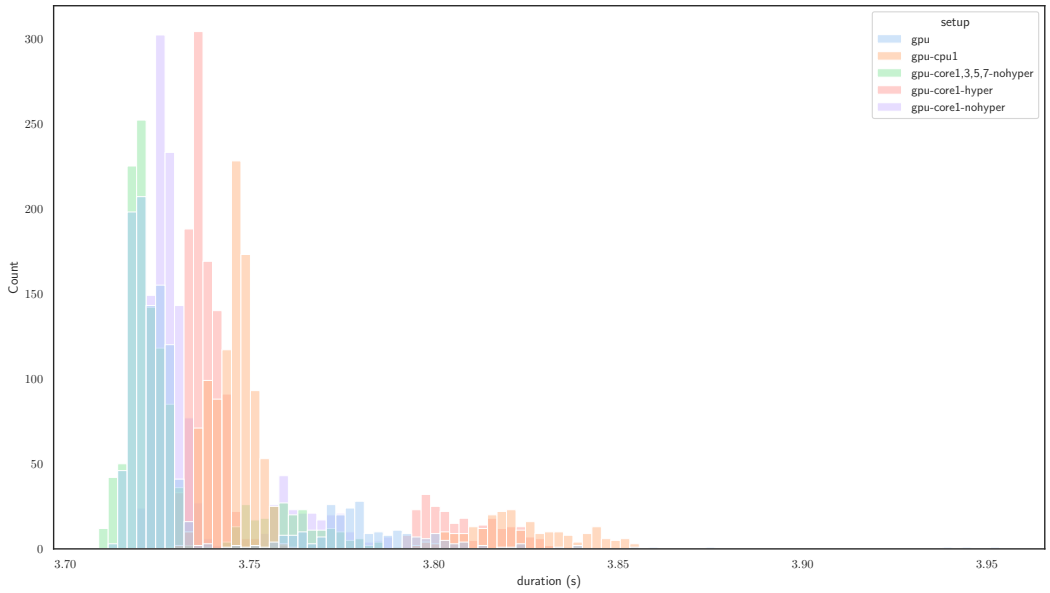


Figure 4.8: Distribution of the durations of each epoch.

4.3.2 Python for Web Development

Django[8] and Flask[9] are the most popular Pyhton frameworks for web development. According to a Jetbrains poll,[10] 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.

Life-cycle of a request in Django Django is an MVT (*Model-View-Template*) framework, which means that it follows the MVC (*Model-View-Controller*) pattern. Figure **??** 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 these layers process the request, 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

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

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 an MVT model, it also offers an automatic way to retrieve data from the database to the view using the *Object Relational Mapping* (ORM) [**?**].

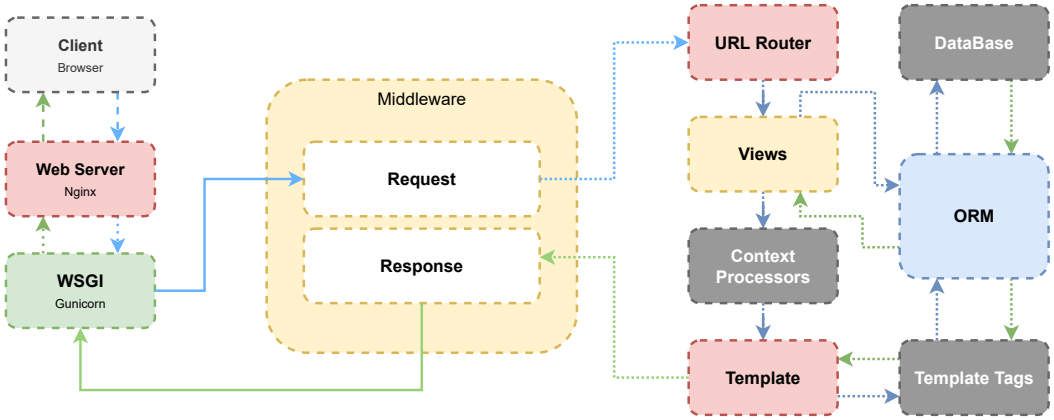


Figure 4.9: Request-Response life cycle in Django

First, we investigate 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 to the request. We tracked its energy consumption using JouleHunter.[11] JouleHunter is an open-source library that we developed to help practitioners identify energy hotspots in their applications using statistical profilers. In the case of Django, JouleHunter is included as a middleware 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 **??**. As we can notice, 91.4% of the total energy consumption is spent on resolving the request by retrieving the data, while only 5.27% is consumed on rendering the response.

Therefore, we chose to study if the choice of the database and the ORM impact energy consumption and web application performance. 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,
2. Prefetch queries the data before being requested,
3. Optimized leverages SQL without passing by the ORM.

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

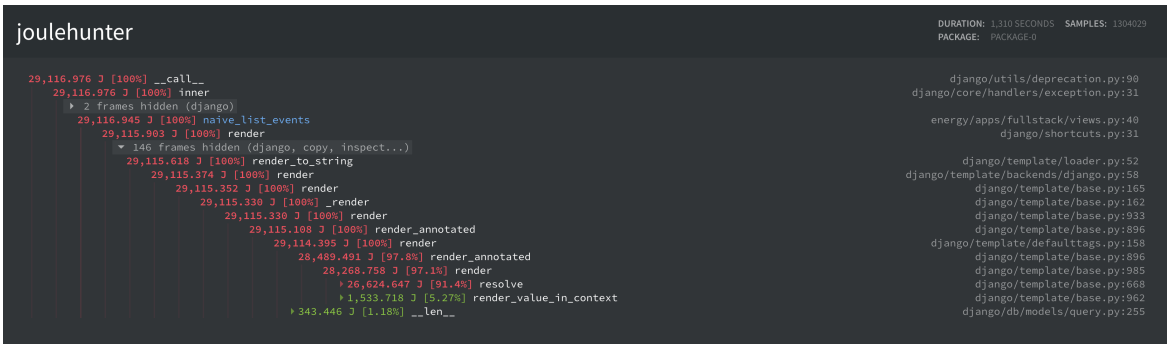


Figure 4.10: Tree representation of the energy consumption of a single request in Django (naive version).

As one can observe in **??**, the strategy to query stored data greatly impacts 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 **??** highlights. For example, using the PyPy interpreter reduces energy consumption, even when adopting the Vanilla strategy. We will further discuss the choice of Python runtime implementation in the next section.

Finally, we run the same experiment with the Optimized strategy using JouleHunter. Figure **??** 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×.

Synthesis We can conclude that the database and ORM selection significantly impact the web application performance and energy consumption. Moreover, unlike rendering responses, many alternatives can be considered to store and query the application database.

4.3.3 Python for Data Processing

Another field of investigation is the type of data and control structures that might impact energy consumption. In this study, we will run some common algorithms with different data structures and control structures to study if there is any difference in energy consumption.

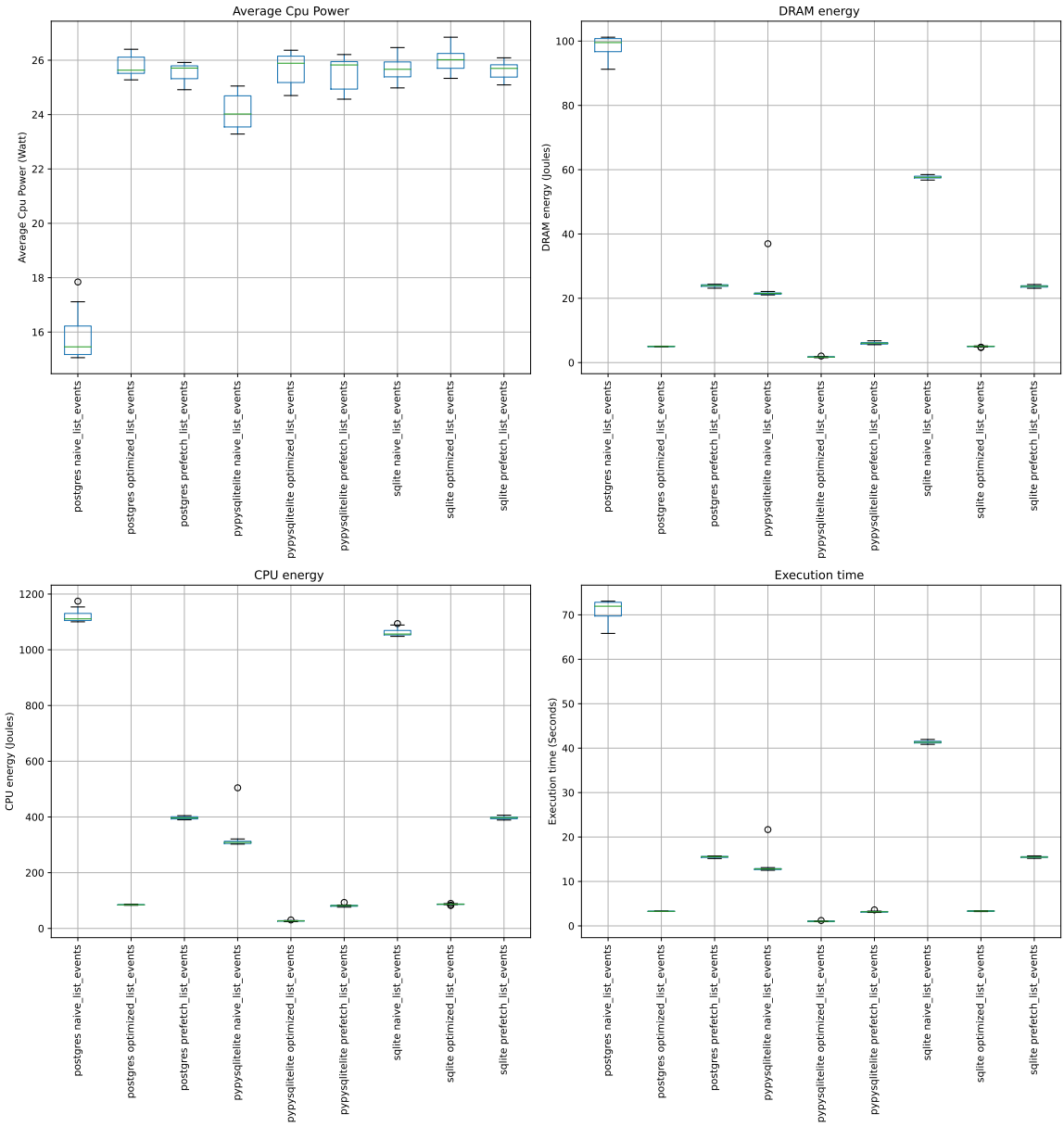


Figure 4.11: Key performance indicators observed for different data access strategies.

Experimental Protocol

All of the tests in this study are done on the Grid5000 cluster using the same machine, Python interpreter, and library version. Due to PowerAPI's frequency constraint, we measure the energy of $1,000$ algorithm iterations for each test. Furthermore, we perform each test 20 times for completeness, deleting the cache between each run. The data is then averaged and presented in graphs.

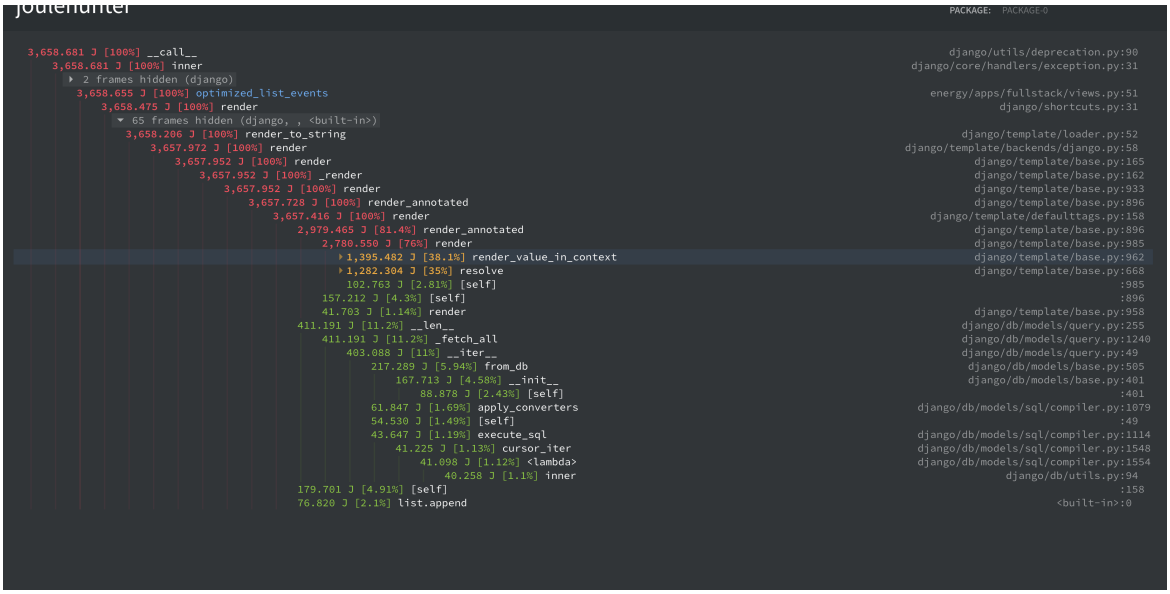


Figure 4.12: Tree representation of the energy consumption of a request in Django (optimized version)

Python Loops

In the first experiment, we study the impact of the type of loop on energy consumption. To do so, we execute the following code snippet:

```
1: sum ← 0
2: for i ← 0 to N do
3:         sum ← sum + 1
4: end for
5: return sum
```

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 collection length and then using the iterator range. So, we tried an alternative version: `for(element in collection)`. Moreover, in most programming languages, the `for` loop is translated into a `while` loop, so we wanted to compare this with a `while` version. Therefore, our candidate loops are defined as:

- `for (i in range(len(n)))`,
- `for (element in collection)`,
- `while`.

Furthermore, we run the same code snippet for each of these implementations with different primitive data types: `Int`, `Float`, and `String`.

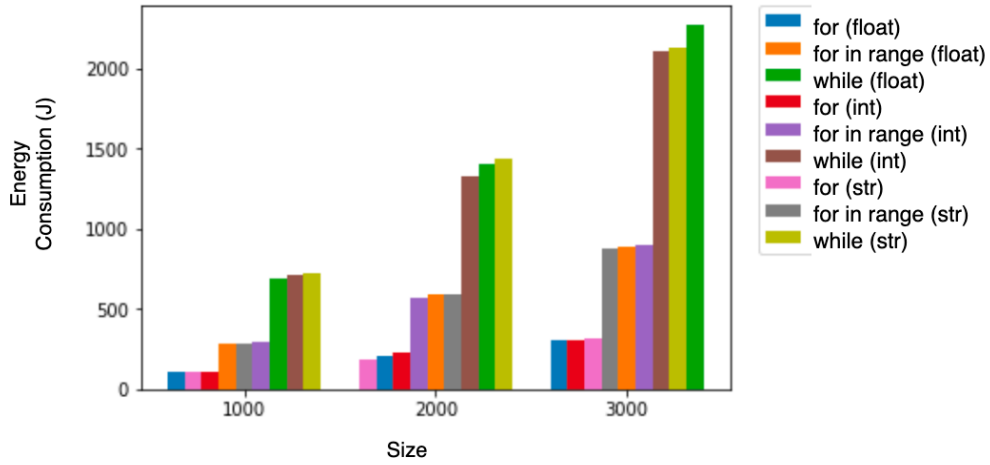


Figure 4.13: Comparison of the energy consumption of different Python loops.

Figure **??** shows the results of the experiment. As one can notice, the data type has a negligible impact on 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.

The reason behind such behavior is mainly related to how the Python interpreter is implemented.[12],[13] Most of the built-in functions and operations are written in C, to reduce the latency of Python applications, and the same goes for the function `range`.[14] 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 [15]. 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. **??**). We used four different methods to do so: `comprehension list`, `while loop`, `a map with predefined function`, and `a map with an anonymous function`. A comprehension list is a Pythonic way to create a new list by applying a function to each

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

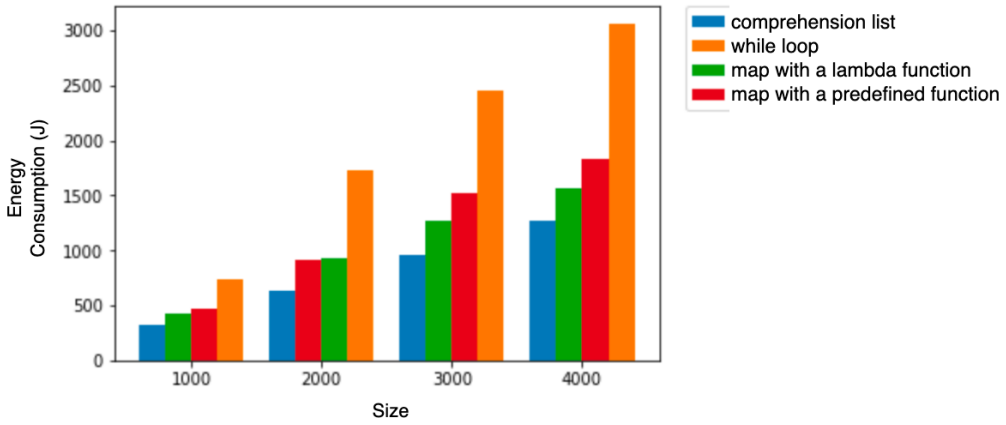


Figure 4.14: Comparison of the energy consumption of different methods to convert a list.

element of the previous one. It is based on the mathematical set-builder notation [**?**]. The map function is a higher-order function that applies a given function to each collection element. It is also possible to use an anonymous function, which is a function that is not bound to a name but only to its definition, which in the Python case is called a lambda function/expression.

As one can notice in **??**, the built-in methods are the most energy-saving ones, while the customize `while` loop is the heaviest. Moreover, when increasing the collection size, the map with lambda functions tends to consume less energy than the predefined one. The reason behind such behavior 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.

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

Python & Multiprocessing

The purpose of this part is to study the impact of concurrency on energy consumption for Python applications. To do so, we run a simple code snippet that computes the sum of the first N numbers using the standard concurrency libraries `multiprocessing` [16] and `multihtreading` [17]. We run the following strategies using a Desktop machine with Intel(R) Core(TM) i7-4800MQ CPU @ 2.70GHz with four cores and four hyper-threads.

[16]https://docs.python.org/3/library/multiprocessing.html
[17]https://docs.python.org/3/library/threading.html

- Sequential: we compute the sum of the first N numbers with no concurency;
- Multithreading: we use the `ThreadPoolExecutor` library to compute the sum of the first N numbers. We divide N by the number of threads, and each thread will compute the sum of the numbers in its range. For our case, we used four threads;
- Multiprocessing: we use the `ProcessPoolExecutor` library to compute the sum of the first N numbers. We divide N by the number of processes, and each process sums the numbers in its range. In this situation, we considered 2, 4, 8, and 16 processes iteratively.

Figure **??** shows the results of the experiment regarding four metrics: power, execution time, DRAM energy, and CPU energy. As one can notice, CPU energy is ten folds higher than the DRAM one. Therefore, we will focus on the CPU for this study.

In general, the MULTIPROCESSING strategy is the most energy efficient one, and the fastest one. However, unexpectedly, the Multithreading option took more time to execute the code snippet than the Sequential one. Therefore, we will divide our analysis into two parts: the first one will focus on Multithreading versus the Sequential strategies, while the second one will study the behavior of the MULTIPROCESSING strategy.

Multithreading vs Sequential The unexpected behavior of the MULTIPROCESSING strategy is because the Python interpreter is not thread-safe. Because of the *global lock system* (GLS), Python cannot run multiple threads at the same time. Therefore, the Multithreading strategy is slower than the Sequential one. On the other hand, a side effect of this is the context switching between threads, which will put the CPU in a lower power state. The Multithreading strategy depicts an average power of $13Watts$ compared to the Sequential one, which is around $22Watts$. This difference in power consumption has overcome the lack of performance of the Multithreading strategy, which leads to better energy efficiency.

Multiprocessing Unlike the Multithreading strategy, the MULTIPROCESSING strategy is based on forking the program into multiple processes, which are independent of each other. This allows the Python interpreter to run multiple processes simultaneously, hence explaining the better performance of the multi-processing strategy. However, this will increase the average power consumption of the CPU, which is around $40Watts$, compared to the Sequential one, which is around $22Watts$. Nevertheless, the MULTIPROCESSING strategy is still the most energy-efficient one.

Although increasing the number of processes will reduce the execution time because we split the tasks on the number of processes, there is a point where the execution time increases.

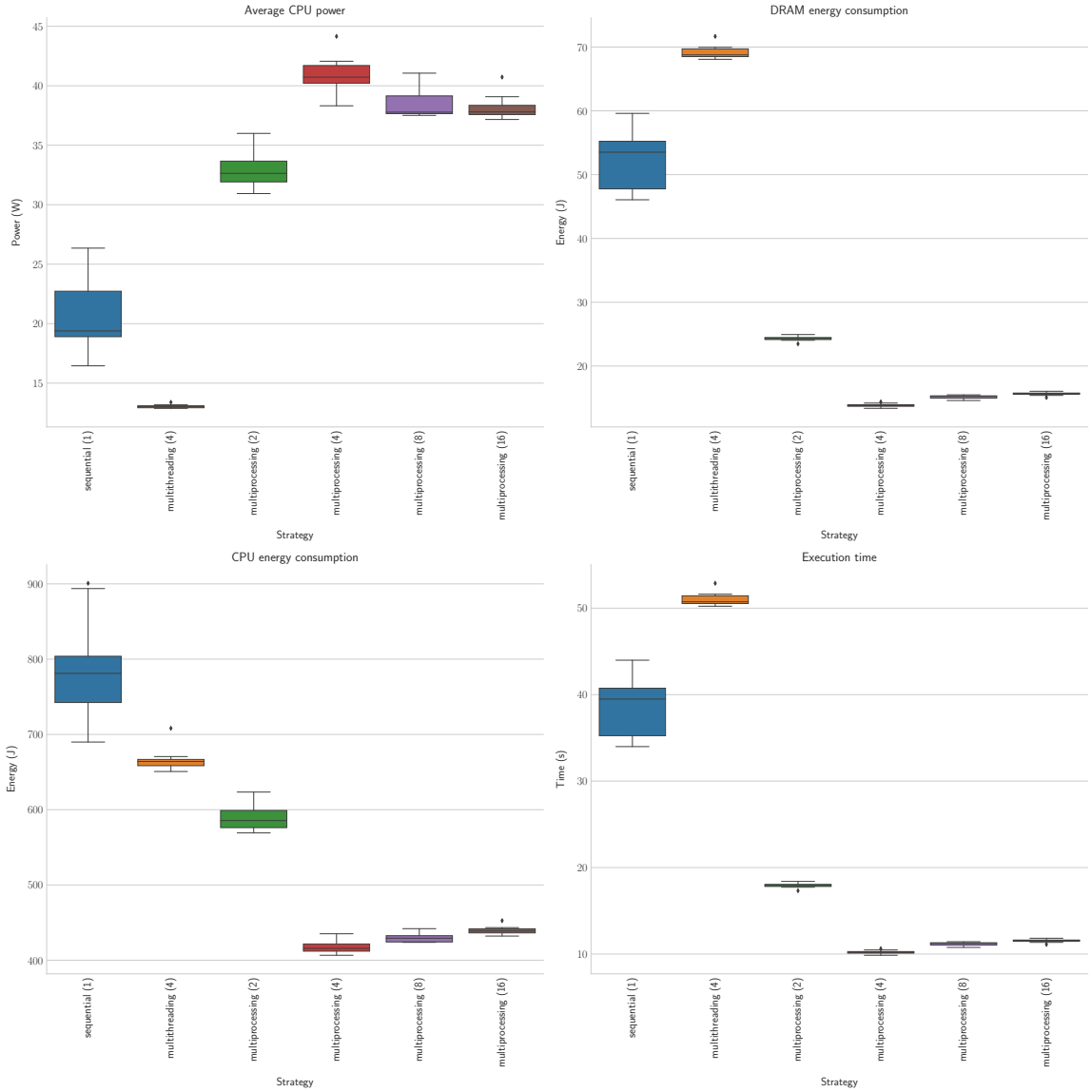


Figure 4.15: Energy consumption resulting from different Python concurrency strategies (number of workers).

As one can see in **??**, the energy consumption correlates with the number of processes until the limit of physical cores, when concurrent processes compete for the CPU.

Before reaching the limit of physical cores, we also observe that the operating system's scheduler 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 by leveraging the ACPI P-states and C-states of unallocated cores, this leads to increased execution times.

Synthesis This study demonstrates that the optimal way to reduce the energy consumption of Python applications is to adopt the MULTIPROCESSING strategy, which is the most energy-efficient one. However, this strategy should be used carefully as it can lead to increased execution times. This study demonstrates that the optimal number of processes is equal to the number of physical cores, which is four in our case. When using the Multithreading strategy, it was also found that sometimes the performance of the application has to be given up to save energy.

4.4 Python Interpreters

We used the official version of Python for the first studies because the goal was mainly to highlight the code structure's impact 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. Despite the availability of numerous tools for code refactoring and automatic code optimization, the optimization process remains a manual effort. This is because code optimization is a task that requires extensive understanding of both the codebase and the language itself.

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 to many initiatives to implement their interpreter to improve one or many aspects of the Python code. Therefore, in this section, we will examine at how different Python interpreters affect the energy consumption of software, and in which ca se, one should use a non-conventional interpreter to save the energy consumption of their application.

4.4.1 Python Runtimes

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

1. **CPython**:[18] 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**:[19] 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 is only applied to the code written by the developer and not to external libraries;

3. **Cython**:[20] 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 translate Python objects into native types, such as integers, and

[18]https://www.python.org/
[19]http://pypy.org
[20]https://github.com/cython/cython

float. This often means better performances since native C libraries are almost all the time faster than the Python written once [**?**];

4. **Intel Python**:[21] A customized interpreter developed by Intel to enhance the performance of Python programs. It is dedicated to data sciences and high-performance computing. It uses some Intel kernel libraries, such as Math Kernel Library (Intel MKL[22]) and data analytics acceleration library (Intel DAAL[23]). It supports both versions of Python;

5. **Active Python**:[24] 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**:[25] A .Net-based Python interpretation platform is written in C# and 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**:[26] A Python interpreter that is based on GraalVM[27] (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**:[28] 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 Python performances;

9. **MicroPython**:[29] A lightweight Python version dedicated to embedded systems and micro-controllers;

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

10. **Nuitka**:[30] A Python compiler is 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**:[31] A library that includes a JIT compiler to enhance the performances of Python functions using the industry-standard LLVM compiler library;

12. **Shedskin**:[32] A static transpiler that translates implicitly statically typed python into C++ code;

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

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

15. **Stackless Python**:[33] An interpreter that focuses on enhancing multi-threading programming;

16. **Pyjion**:[34] A JIT API for CPython, same purpose as Parakeet and Hope;

17. **Pyston**:[35] A performance-oriented Python implementation built using LLVM and modern JIT techniques. The project is funded by Dropbox;

18. **Grumpy**:[36] 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 (the last commit was in 2017), and it has many limitations in terms of supporting the Python language, such as some built-in functions and standard libraries;

19. **Psyco**:[37] A JIT compiler for Python;

20. **Unladen Swallow**:[38] An attempt to (use) LLVM as a JIT compiler for CPython.

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

4.4.2 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 runtime infrastructure category that supports a Python application's execution. 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 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.

We did not consider other implementations because either the project aborted many years ago or it has insufficient Python features. After the inventory 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 three categories depending on their integration with the Python code. In **??**, we describe the implementations we kept, each implementation's version, and its category.

4.4.3 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 used to answer the research questions addressed in this manuscript.

Table 4.2: 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	-	–	✓	–

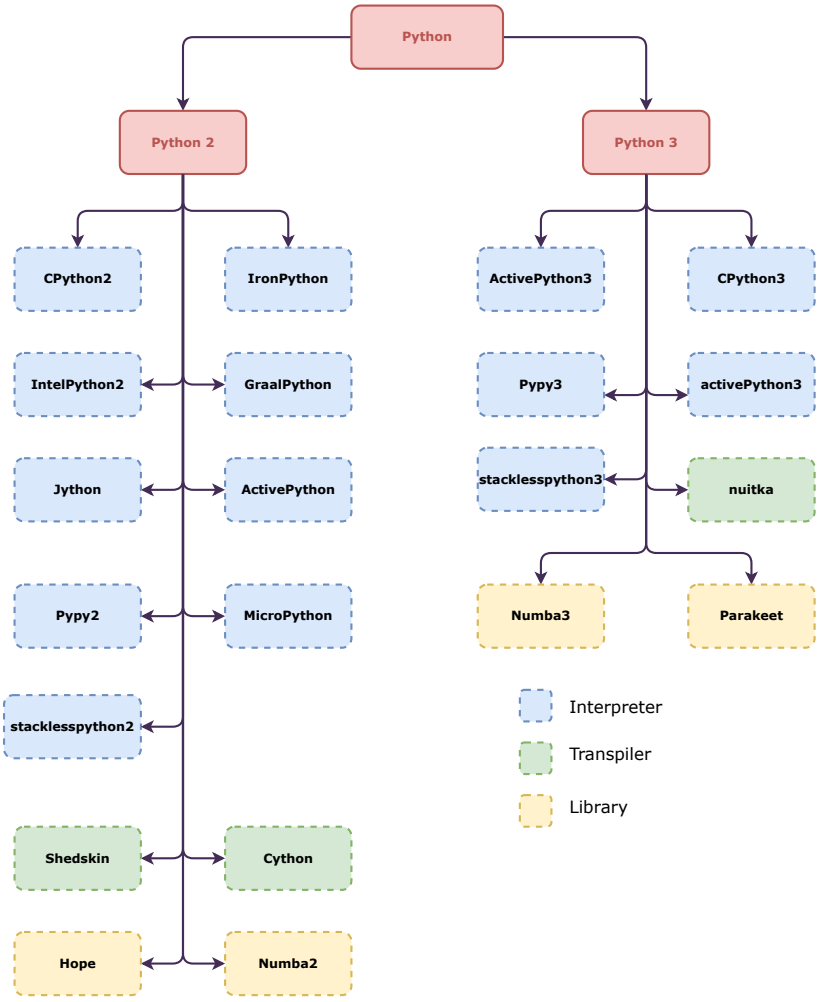


Figure 4.16: Python interpeters

Table 4.3: 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

Measurement Context

Hardware settings. All our benchmarks have been executed on a Dell PowerEdge C6420 server, whose hardware features are summarized in Table **??**. 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 4.4: Benchmarking server configuration.

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 correlates highly with the energy, and it will be only useful to explain some specific energy behaviors. Thus we do not put much focus on this metric.

Energy measurement. As we know, a program's energy is integral to its power over time. For our case, we used Intel *Running Power Average Limit* (RAPL) [**?**] to collect the power

samples of the running tests. We run POWERAPI [**?**], 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:

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

Figure **??** overviews the architecture of our benchmarking infrastructure.

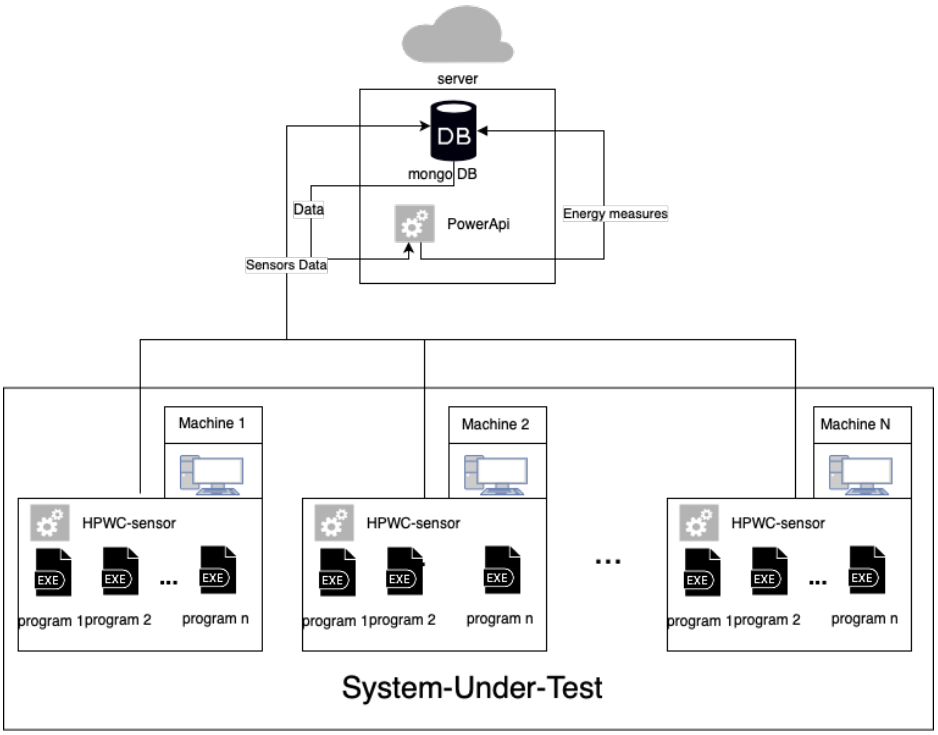


Figure 4.17: Benchmarking architecture deployed with POWERAPI.

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. We employed the TOMMTI microbenchmarks suite to benchmark our implementations.[39] TOMMTI is a set of 13 microbenchmarks that examine common language features, such as arithmetic operations, data structures, and input/output manipulations,

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

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 work 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 effects, 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 Python3 does not support it, we converted the code using the official library *2to3*.[40] In the case of the libraries using *JIT*, adding a decorator to the function 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 [**?**] 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 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,[41] we propose a dedicated tool to generate new workloads and new candidates. The script `generator.py` allows practitioners to create new benchmarks by implementing 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 put in a recap file called

[40] https://docs.python.org/3.7/library/2to3.html
[41] https://github.com/chakib-belgaid/python-implementations

`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. Suppose extra manipulation should be performed on the workload files, such as adding a new decorator or changing some parameters. In that case, they should be added as an extra function in the script `generator.py`. Finally, they should be included in the Python candidates.

4.4.4 Results & Findings

This part will be dedicated to the results of the experiments and statistical analysis of these results. First and foremost, we want to know whether our samples have a normal distribution. As a result, we ran a Shapiro-Wilk [**?**] test on our data. Even though 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 rejecting 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 [**?**], with CPython2 and CPython3 as base references.

The Mann-Whitney U test results for each implementation are shown in Table **??**. The first column shows the average energy usage of the implementation, while the second shows 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, indicating that the energy consumption of the given implementation differs significantly from that of the reference implementation. As can be seen, the majority of the implementations deviate significantly from the reference implementation.

As we want to study the position of these implementations with regards to the reference implementation (aka Cpython2 and Cpython3), we proceed with agglomerative *hierarchical cluster analysis* (HCA) [**?**]. 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 **??** displays the Dendrogram produced by the HCA.

Except for GraalPython, one can notice three 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

Table 4.5: 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

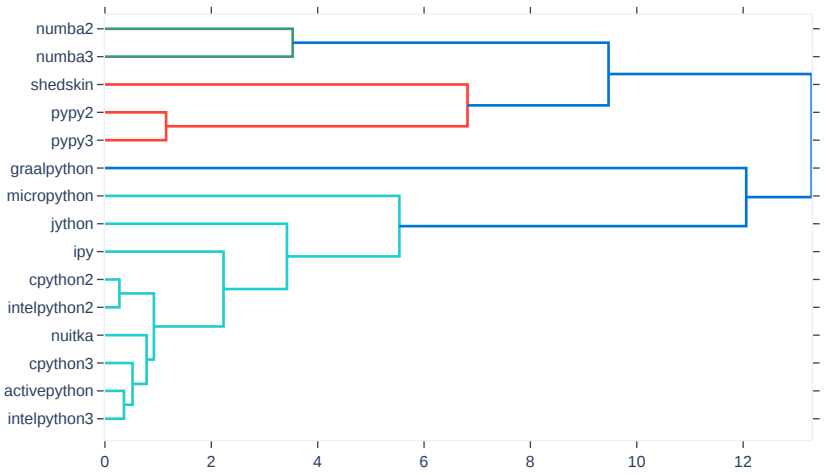


Figure 4.18: Dendogram of the different Python runtime implementations.

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 to be later compiled into binary code. Table **??** 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 decides the code parts to optimize, Numba gives this responsibility to the developer.

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 **??** 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 compare numerous implementations due to their large differences.

As one can notice in Figure **??**, 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, not the core interpreter. ActivePython states that its 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.[42] As for IntelPython, it is designed for machine learning. Unfortunately, the TOMMTI benchmark is geared for general-purpose programming. Although 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, looking 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 in arithmetic operations. One can conclude it is due to the native type of the variables, unlike in the JIT, where they are treated as objects initially.

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

Unlike the previous interpreters, GraalPython exhibits a specific promise regarding complex algorithms (nested loops in particular). However, it is still in its early stages of

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

development. It is not yet ready for production since some benchmarks, such as array manipulation and sorting, require an abnormally lengthy time.

Threats to validity

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 different implementations being developed to optimize a specific library, such as Numba with Numpy, or IntelPython with machine learning algorithms. Furthermore, we excluded from this study runtime that could not complete our benchmarks because we judged that this was the bare minimum for an implementation to be employed in a real-world application. Besides this, since the goal of this study was to provide a non-intrusive technique to decrease the energy consumption of software, we opted to omit some of the implementations that required extra work on the source code to run on them, such as Cython.

4.4.5 Summary

One can conclude that the Python interpreter used considerably impacts the programs' energy consumption. Furthermore, the absence of a universal solution makes this investigation more intriguing. While most of the candidates employed Python2, Python3 provided better compatibility coverage. Except for Pypy, a standout interpreter, the optimization strategy substantially impacted energy usage because each class was grouped regardless of the Python version. Finally, JIT, a technology that converts code into machine code at runtime, was responsible for the significant gain in energy efficiency. The usage of translators, such as Shedskin confirmed the effect of compiling Python code into machine code. Hence, PyPy, an interpreter with a built-in JIT, is within the same class as the complied ones, such as Shedskin.

Figure 4.19: Different Python runtime energy scores.

4.5 Conclusion

As many software services use Python in public and private cloud infrastructures, making Python-based apps more energy efficient will lower ICT carbon emissions. This chapter discusses various ways of optimizing the energy consumption of Python-based applications. It first explains why such a choice is relevant. Then, it studies the energy behavior of Python within its most commonly employed use cases, which are revealed to be web development and machine learning.

First, we studied the energy consumption of a machine learning algorithm using a benchmark of 60,000 entries to train the `cifar10-fast` model. We discovered that this energy consumption increases exponentially while the model accuracy increases. As a result, a reduction in accuracy can result in considerable energy savings. We investigated how data structures, parallelism, and iterative methods affect energy use.

As for web development, we looked at a website built with Django, which is one of the most popular web frameworks in the Python community. We found that fetching the data from the database consumes most of the energy during the request processing phase. Therefore, Django-based web servers can use up to ten times less energy if the developer chooses the write strategy to fetch the data.

After that, we analyzed the impact of several iteration mechanisms in Python and their impact on energy consumption. We found that the built-in functions and the best practices for writing Python code are the most energy-efficient ones. This energy efficiency is because most of these built-in functions are written in a lower programming language, C.

Finally, we studied the impact of concurrency on energy consumption for Python applications. We found that the multi-processing strategy is the most energy-efficient one. However, this strategy should be used cautiously. This study demonstrates that the optimal number of processes is equal to the number of physical cores, and exceeding this number will cause the application to loose its efficiency in both performance and energy consumption. As for the multithreading strategy, we found that the Python interpreter is not thread-safe, which leads to a slower performance than the sequential one. However, this also leads to lower power consumption, which sometimes overcomes the lack of performance to make the application more energy-efficient.

In the second section, we presented a non-intrusive technique to optimize the energy consumption of Python-based apps without making substantial changes to the code. This technique consists in guiding the choice of the Python runtime implementation. We started by categorizing and filtering these implementations into three significant classes (compiler, interpreter, and extra libraries). Then, we conducted a series of experiments to compare the energy consumption of these alternatives. Our findings indicate the lack of a general solution

and the importance of tuning the Python runtime depending on applications. We found that most interpreters had a similar or worse energy consumption than the official implementation of CPython. The reason behind such behavior was that some implementations focused on specific case studies, such as machine learning or security. In contrast, others focused on the compatibility of the Python code with their platforms, like Jython and IronPython. Finally, regardless of implementation, we demonstrated that using JIT is the most efficient technique to boost the energy efficiency of Python-based programs.

We believe that our contributions will benefit a broad spectrum of legacy systems, reducing ICT's carbon footprint and lowering cloud bills for these services' resources.

Chapter 5

Impact of Java Virtual Machine Configurations on Energy Consumption

5.1 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.

5.1.1 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 **??**, 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/

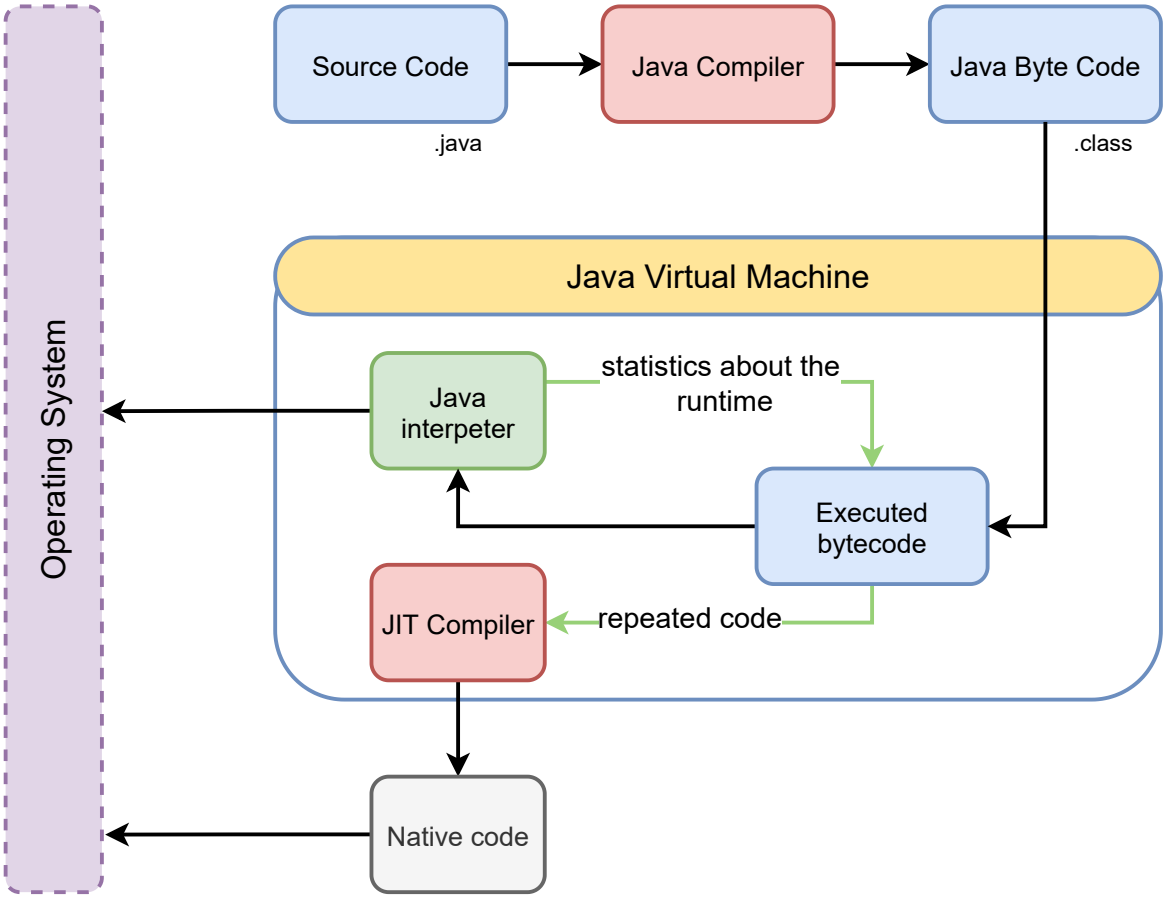


Figure 5.1: JVM architecture

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.

5.1.2 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.

5.2 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.

5.2.1 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 [**?**] 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 5.1: 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 [**?**].

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 **??**. Depending on providers, either all the versions, majors, or LTS are made available by SDKMAN.

5.2.2 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 [**?**], 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 [**? ?**]. 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 [**? ?**], 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. **??** summarizes the selected benchmarks with a short description. Meanwhile, Figure **??** highlights the scope of each benchmark from the test suite.

5.2.3 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 con-

[5]https://openbenchmarking.org

Table 5.2: 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

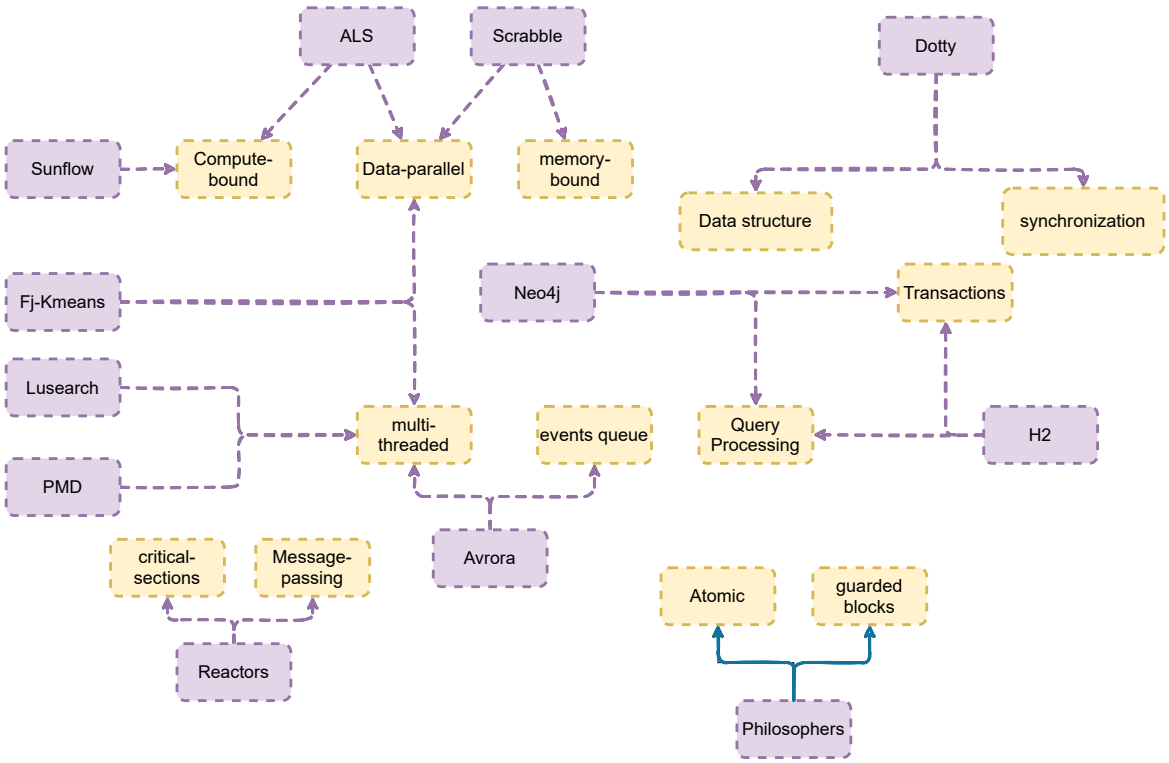


Figure 5.2: Target scope of DACAPO and RENAISSANCE benchmarks.

sumption, 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 [**?** **?**]. We note that, due to CPU energy consumption variations issues [**?**], 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" [**?**]. 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]

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

5.2.4 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 **??**.

5.3 Experiments & Results

5.3.1 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. **??** 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 **??**.

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 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

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

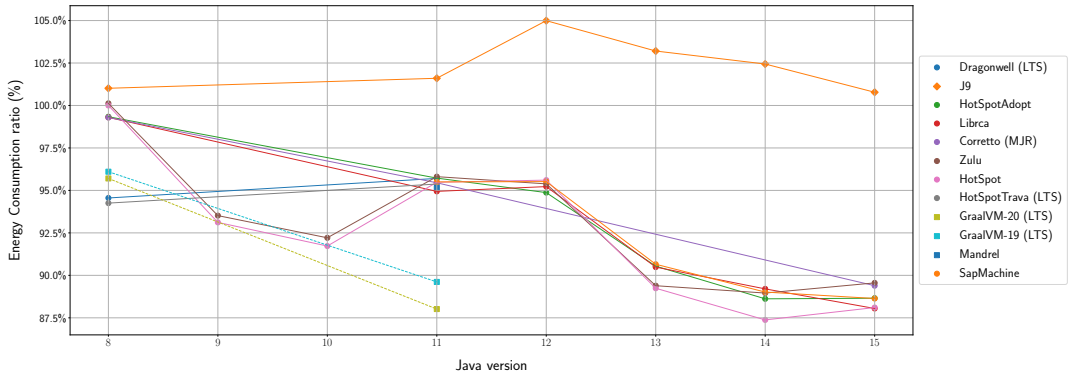


Figure 5.3: Energy consumption evolution of selected JVM distributions along versions.

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 **??**: 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, **??** depicts the evolution of the energy consumption of the 12 benchmarks, when executed on the HOTSPOT JVM. **??** 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 **??**, 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, 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

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

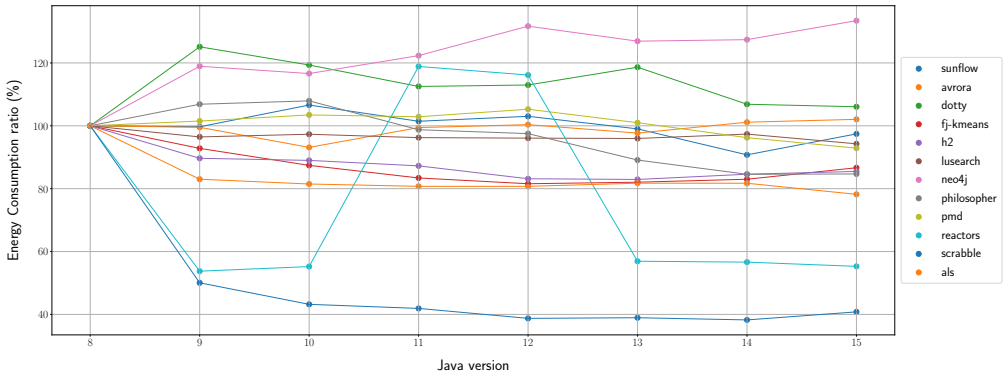


Figure 5.4: Energy consumption of the HotSpot JVM along versions.

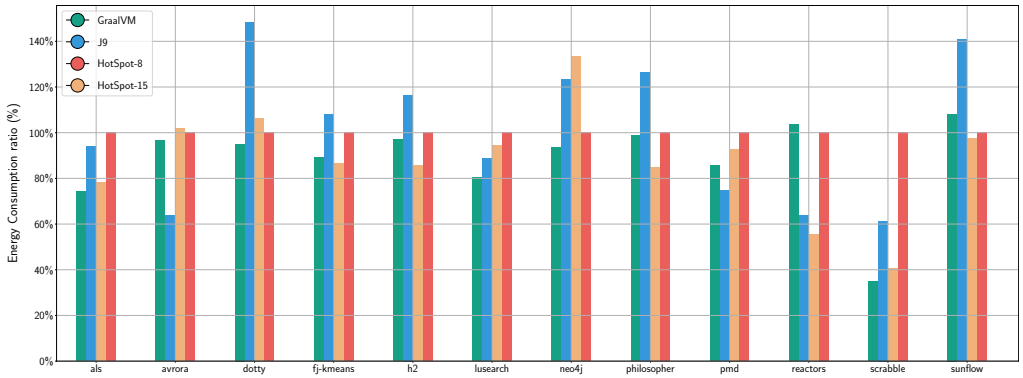


Figure 5.5: Energy consumption comparison across Java benchmarks for HOTSPOT, GRAALVM & J9.

(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.

?? 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

Table 5.3: 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

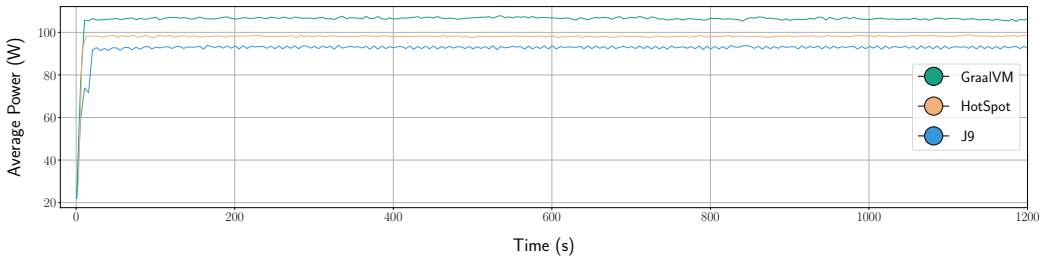


Figure 5.6: Power consumption of Scrabble as a service for HOTSPOT, GRAALVM & J9.

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 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. **??** 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 **??**. 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 **??** for this benchmark.

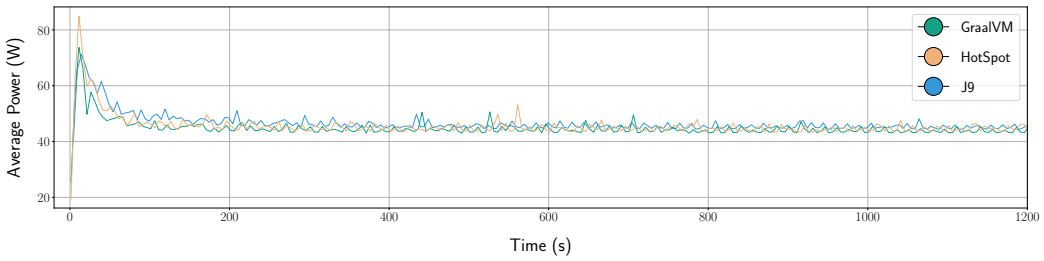


Figure 5.7: Power consumption of Dotty as a service for HOTSPOT, GRAALVM & J9.

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 **??**. 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 **??**; therefore, ignoring the warm-up phase would have been misleading.

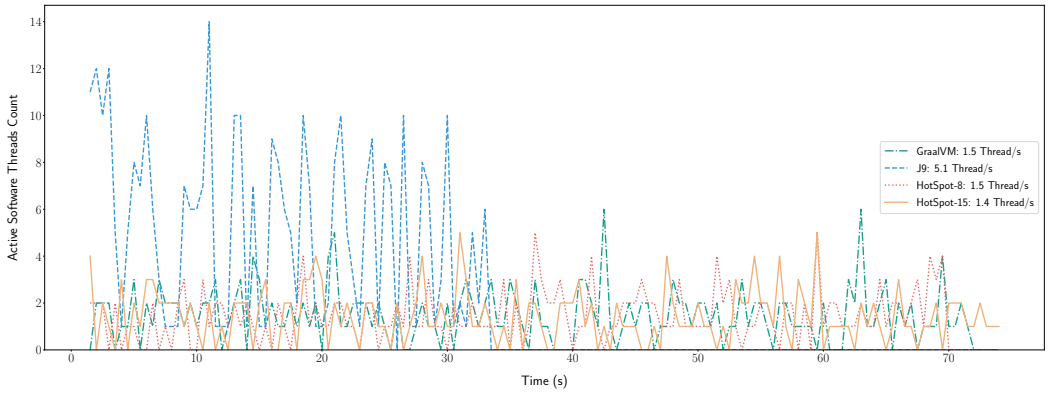
To answer **RQ 1**, we conclude that—while most of the JVM platforms perform similarly—we can cluster JVMs in 3 classes: HOTSPOT, J9, and GRAALVM. The choice of one JVM of these classes can have a major impact on software energy consumption, which strongly depends on the application context. When it 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.

5.3.2 Energy Impact of JVM Settings

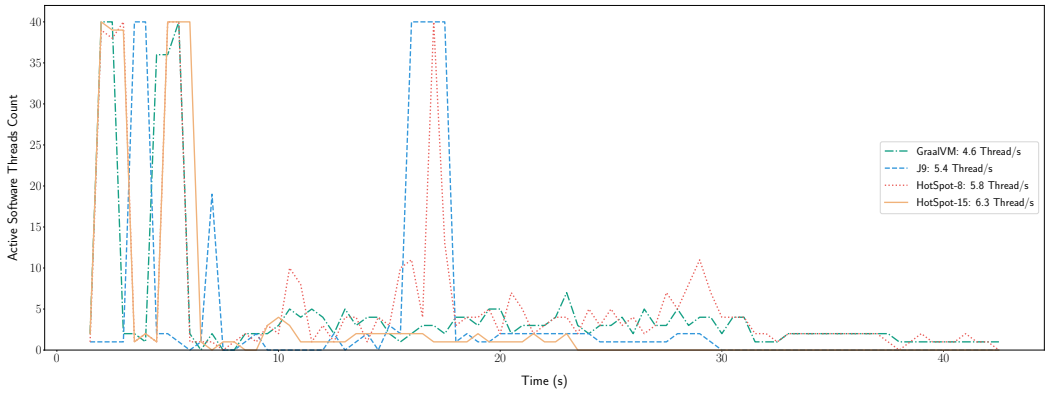
The purpose of our study is not only to investigate the impact of the JVM platform on energy consumption, but also the different JVM parameters and configurations that might have a positive or negative effect, with a focus on 3 available settings: multi-threading, JIT, and GC.

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-



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



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

Figure 5.8: Active threads evolution when using HOTSPOT, GRAALVM, or J9.

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 **??**, namely Avrora and Reactors. We studied their multi-threaded behavior to optimize their energy efficiency.

?? delivers a closer look to the thread allocation strategies adopted by JVM. First, **??** 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.

The second example in **??** 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 **??**, 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

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

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. **??** reports on the energy consumption of the experiments we conducted for most of the benchmarks and JIT configurations under study.

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

Table 5.4: 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**

Table 5.5: 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

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.

Garbage Collection

Changing or tuning the GC strategy has been acknowledged to impact the JVM performances [**?**]. 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 **??**.

For HOTSPOT and GRAALVM, we also considered many GC policies, as described in **??**. 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 5.6: 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

?? 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 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

Table 5.7: 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

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.

5.4 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 [**?** **?**]. 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 [**?**].

Another measurement issue is the CPU energy variation within machines (cf. **??**), 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, 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.

5.5 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.. **??** 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 **??** 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

[13]https://github.com/srikanth-lingala/zip4j

[14]https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/knucleotide.html

Table 5.8: 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

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]

[15]https://github.com/chakib-belgaid/jreferral

5.6 Summary

This chapter describes an empirical investigation of contrast in energy consumption that some of the most well-known and widely supported JVM platforms can exhibit as well as the important factors that can influence this energy consumption. During our experiments, we investigated 12 well-known and diverse-purpose Java benchmarks, as well as 52 JVMs, including many versions of 11 different distributions. Our findings revealed that various JVMs share energy efficiency and may be classified into three major classes: HOTSPOT, J9, and GRAALVM. However, the three JVM classes can show substantially different energy efficiency depending on the workloads and/or application. While there was no clear champion in terms of energy consumption,GRAALVM recorded the highest energy efficiency for the most of benchmarks. Nevertheless, depending on the workload, each JVM can achieve higher or worse efficiency. One explanation could be thread management policy, as seen with J9 when running Avrora.

Additionally, numerous JVM settings can cause changes in energy consumption. Our experiments revealed that the JVM's default JIT compiler is frequently near-optimal in at least 80% of the experiments.

On the other hand , the default GC outperformed the other strategies in just half of our experiments, with some significant benefits reported when employing different GCs based on the application characteristics. Our primary observations and recommendations can be stated as follows: *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 make the above principles easier to implement, we propose *J-Referral*, a tool that recommends the best energy-efficient JVM distribution and configuration from among more than a hundred options. It generates a comprehensive report on the energy consumption of both CPU and DRAM components for each JVM distribution and/or configuration in order to assist the user in selecting the one with the lowest consumption for Java applications.

Chapter 6

Energy Footprint of Distributed Programming Frameworks

Introduction

Nowadays, web applications dominate online systems. From Google to Facebook, web applications are widely deployed across organizations and continuously accessed by end-users for personal and professional daily tasks. In practice, the development of these web applications heavily relies on a vast ecosystem of *web frameworks*, intended to ease and foster the development process.

However, once deployed, the applications developed with such web frameworks do not exhibit the same performance as reported by the *Web Framework Benchmarks* peri odically published by the TECHEMPOWER company [1].

Thanks to such benchmarks, developers can make informed decisions on the most efficient technology to adopt to implement their web applications.

Unfortunately, when selecting a web framework, developers and benchmark providers focus primarily on popularity and performance criteria, with little regard for the resource consumption implications of their choice. This is especially unfortunate, given that developers increasingly turn to cloud providers to host their web applications. In such a context, the energy consumption of web applications is a critical factor, as it directly impacts the cost of hosting the applications. While cloud providers provide a convenient elastic provision of resources to scale based on application requirements, this convenience may come at a high cost to their business. Indeed, the energy consumption of cloud data centers is a primary concern for cloud providers, as it represents a significant portion of their operational costs.

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

Beyond the economic 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 few years, thus causing a rebound effect on the power consumption of server infrastructure hosted or supported by cloud providers.

While one can challenge the relevance of features that developers continuously deploy to keep engaging end-users, reconciling economic and environmental concerns remains an open challenge to address.

Given this context, this chapter 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 from ICT.

To achieve this objective, we consider two study cases. First, we study the impact of programming languages while handling requests from the Remote Procedure Call protocol (RPC). RPC is a protocol that allows a computer to call a procedure stored in another address space (usually on another computer on a shared network) without the user having to explicitly code the details for this remote interaction. We compare multiple implementations of GRPC library, a new RPC library developed by Google and based on PROTOCOL BUFFERS, a popular serialization protocol. As for section **??**, we study the impact of web framework stacks on energy consumption. To do so, we implement a simple web application using the most popular web frameworks and compare their performance, latency, and energy consumption. We leverage the TECHEMPOWER *Web Framework Benchmarks* to incorporate server-side energy measurements obtained from a software-defined power meter, named POWERAPI [**?**]. These measurements are then analyzed in depth to understand the critical criteria that can impact the power consumption of web frameworks and derive guidelines for supporting developers in picking the most energy-efficient web frameworks according to their requirements.

6.1 Investigating Remote Procedure Call Frameworks

With the success of the Internet and the emergence of cloud technologies, many communication protocols compete to take the lead. In particular, most software architectures are

now based on multi-services and micro-service technologies. Moreover, to cope with the increased versatility of developers, multiple companies choose to open their services to different programming languages. This approach is interesting because it wants to use the best parts of each programming language to meet different needs. However, the challenge nowadays is to make the bridge between these 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 The Remote Procedure Call (aka RPC) protocol. In order to provide a universal implementation of this protocol, Google developed GRPC, a contemporary open-source RPC framework that may run in any environment. Pluggable load balancing, tracing, health monitoring, and authentication support may efficiently link services within and across data centers. It is also valuable for the final mile of distributed computing, connecting devices, mobile apps, and browsers to backend services.

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 2: How do RPC implementations consume energy regarding the number of concurrent clients?

RQ 1: How do RPC implementations consume energy concerning the size of the incoming request?

6.1.1 Experimental Protocol

To answer these questions, we will define an interface using the Proto2[2] language, and then we will implement it in multiple servers, each one using a different programming language. We will then compare the energy consumption of each implementation using a custom version of `GHZ` [3], a tool that allows us to stress test our servers. This fork of `GHZ` allows us to measure the server's energy consumption during the test's execution and synchronize it with other performance statistics such as Tail latency and the number of requests per second the server can reach. Figure **??** show the experimental protocol. We will use different messages inserted in the file `helloworld.proto`. The rest of this part will be dedicated to explaining each aspect of the experiment.

[2]https://developers.google.com/protocol-buffers
[3]https://github.com/chakib-belgaid/energy_ghz

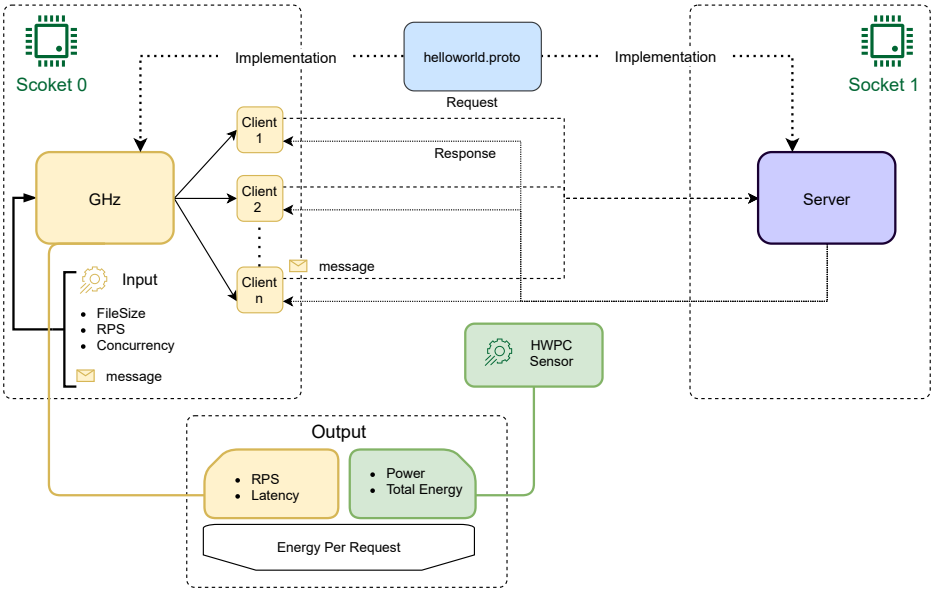


Figure 6.1: Experimental software architecture.

Measurement Context

Hardware settings. All the experiments are run on the cluster paravance of the G5K platform. This cluster comprises 72 identical machines 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 networks' impact 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 energy consumption, we used the `HWPC` sensor[4], based on Intel `RAPL` technology, one of the most accurate tools to measure the CPU's and DRAM's energy consumption [**? ?**]. 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 [**?**].

Performances. For better accuracy and more details, We updated the open source RPC benchmarking tool 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.[5]

Input Workload

The purpose of the experiment is to analyze the behavior of different GRPC implementations. Therefore, we have two kinds of workloads: *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, allowing 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.[6] 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** and by changing the number of clients in the benchmarking tool.

[4]https://github.com/powerapi-ng/hwpc-sensor
[5]https://github.com/chakib-belgaid/energy_ghz
[6]https://github.com/chakib-belgaid/energy_benchmarking_grpc2

6.1.2 Results & Findings

Preparing the Data Before we start our Analysis, we will first clean the data from the outliers using the interquartile range (IQR) method [**?**]. For each implementation, we will calculate the Q3 and Q1 of the population; then, we will exclude all the values outside the range $[Q1 - 1.5 \times IQR, Q3 + 1.5 \times IQR]$.

[RQ 1:] How do RPC implementations react to the number of clients?

In this part, we will analyze the behavior of the different implementations regarding the number of clients.

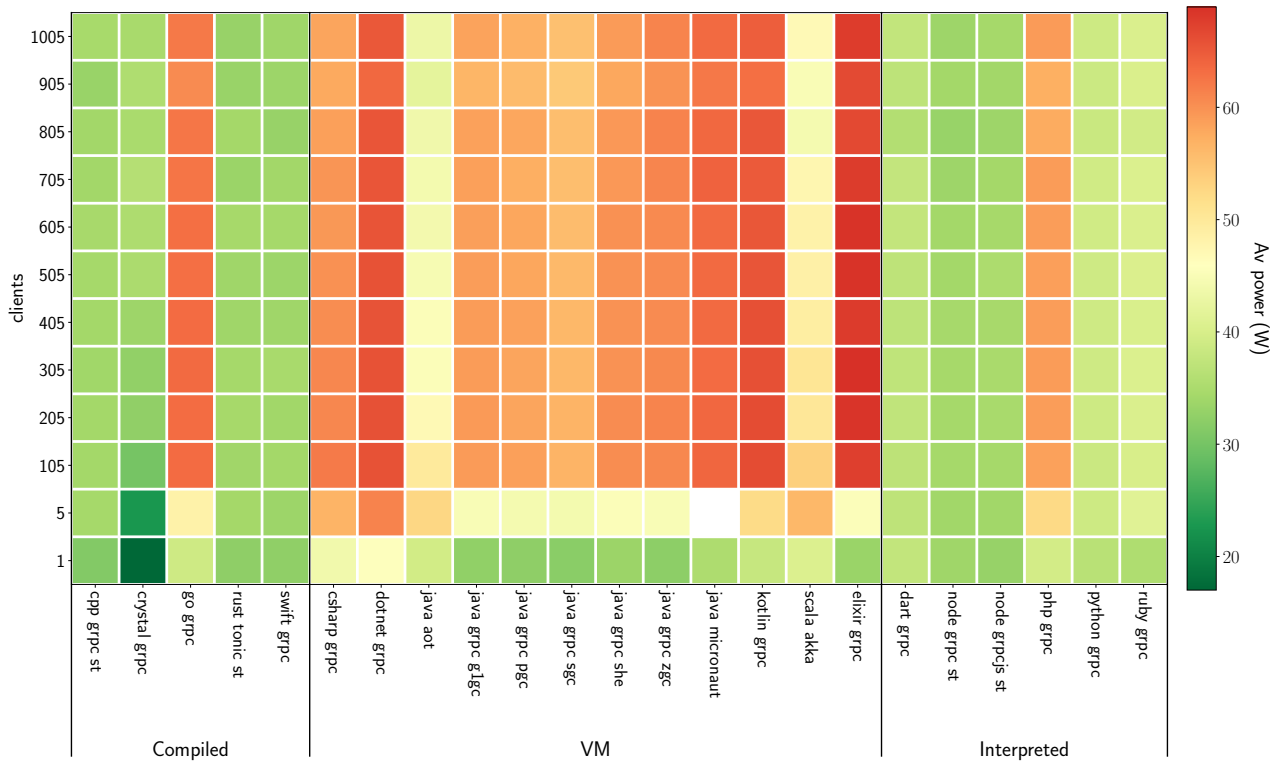


Figure 6.2: Average power consumption based on the number of the clients

Power behavior First, we start with the overall power consumption for each server implementation. Figure **??** shows a heatmap of the average power consumption of each implementation. As one can notice, there are two main scenarios. The first one is when the number of clients is lower than 100, which we will call lite scenario, and the second one, the stressed scenario, is when the number of clients is more exceeds 100.

Lite scenario 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 45 Watts.

In each programming category, we observe both energy-efficient and energy-greedy behaviors. Therefore, we conclude that it depends more on the library's implementation than the programming language category. Scala and Kotlin are excellent examples to support this hypothesis, as both run on the same virtual machine as Java (OpenJDK 16.1). However, their average power is 130% higher than the Java implementation.

Stressed scenario Although the same classes remained the same, not all languages had the same evolution. Here we can observe a correlation with the category of the programming language rather than the implementation itself. We can quickly point out that the average amount of power used by VM-based languages doubles when they have more than 100 clients simultaneously. Except for PHP, all the interpreted and compiled languages preserved their energetic behavior. Our hypothesis points to the JIT since it compiles the code and makes it run faster, hence stressing the CPU. An interesting behavior has been noticed for the GraalVM: decreasing energy consumption when increasing the number of clients. This is related to the performance drop, probably due to the bottleneck situation where the GraalVM could not handle more than 100 clients simultaneously.

Performance Behaviour this paragraph will cover the number of requests per second processed per server without looking at its energy. Here, 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 reducing the number of requests per second or increasing the processing time. However, some frameworks, such as Dart or Scala, have taken a different approach, maintaining a specific latency limit even if not all requests are answered. Furthermore, we tend to observe this behavior among other frameworks, such as Python or Asynchronous NodeJS, when the number of clients exceeds 800. To dive more into this behavior, the reader can refer to the following GitHub repository https://github.com/chakib-belgaid/grpc_analysis

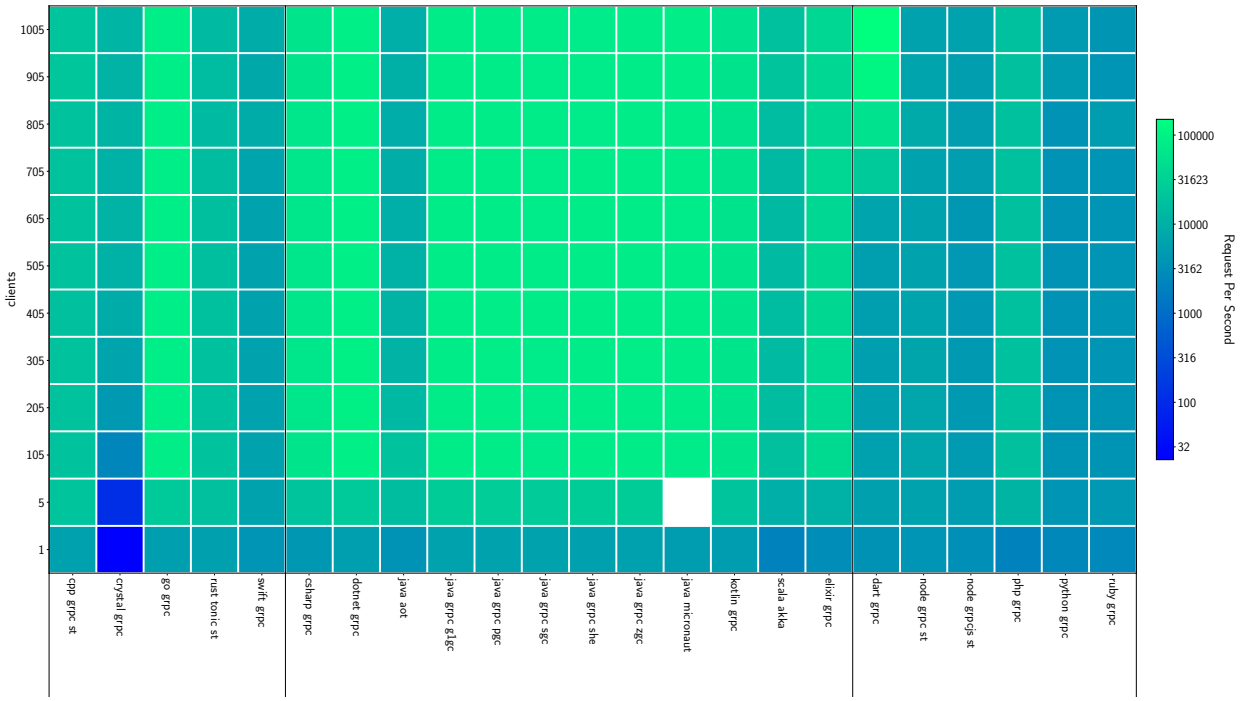


Figure 6.3: Number of requests per second based on the number of clients

RPS Figure **??** presents the number of requests per second for each implementation. The heatmap is colored in a logarithmic scale to visualize the results better. As one can observe, most of the servers hit their RPS limit after five clients and 100 clients for VM-based servers, and after this, the number keeps constant, decreasing 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 Even if the number of requests per second increases, that does not mean the latency will go down. As one can notice in figure **??**, until the 1000 clients, Go provides the least latency besides.Net. GraalVM provides the highest average latency. However, Dart tends to become slower when we increase the number of clients until we pass the 600 simultaneous clients, and then it changes its behavior. Instead of satisfying most of the requests, it notifies the clients directly that the server is saturated, resulting in a drop in satisfaction ratio and an amelioration in average latency.

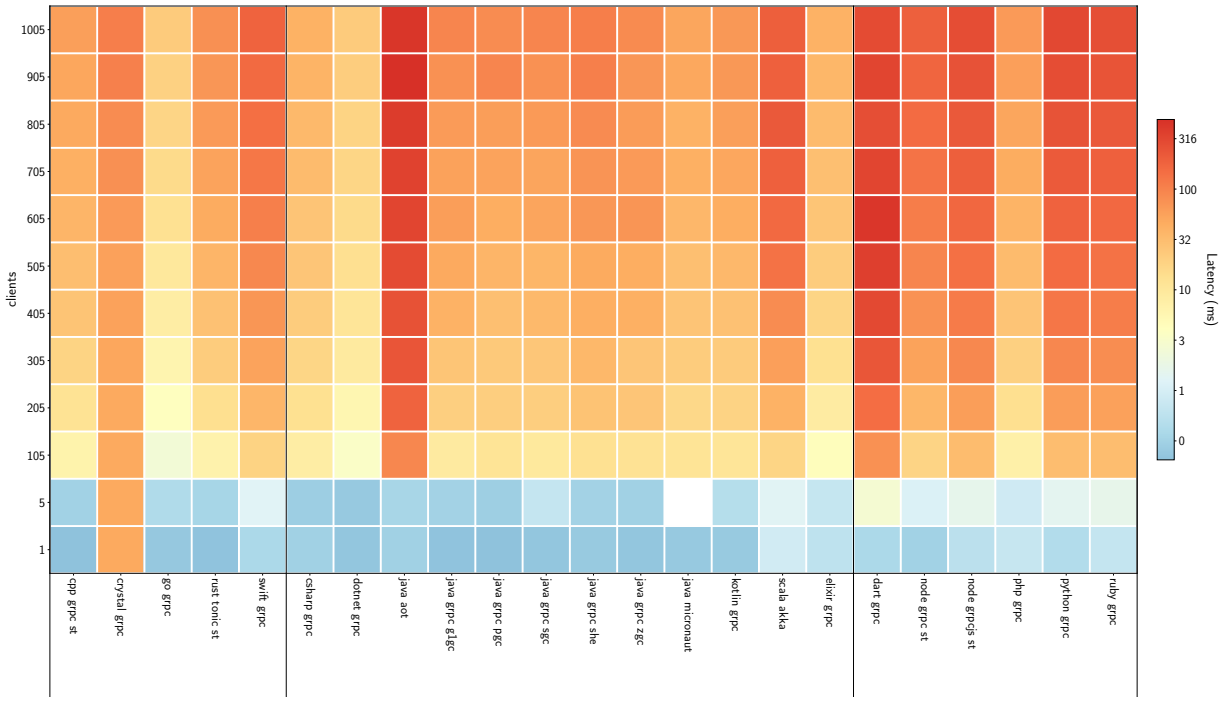


Figure 6.4: Tail latency (99%) based on the number of clients

Energy Per Request After separating the energy and the performances, we have seen that most performance servers tend to be energy-greedy, so we propose investigating this trade-off between energy and performance. To do so, we report the average cost of a single request in millijoules in Figure **??**.

Except for GraalVM (aka Java AOT), the cost of the single request decreases when we add more clients.Java, .Net, and go are the most energy efficient, while Python and Ruby may cost up to 10x more. Despite its low power, textsfCrystal exhibits greedy behavior when dealing with requests from fewer clients. Such behavior is caused by the low number of requests per second, which also results in high latency. Therefore, we conclude that the

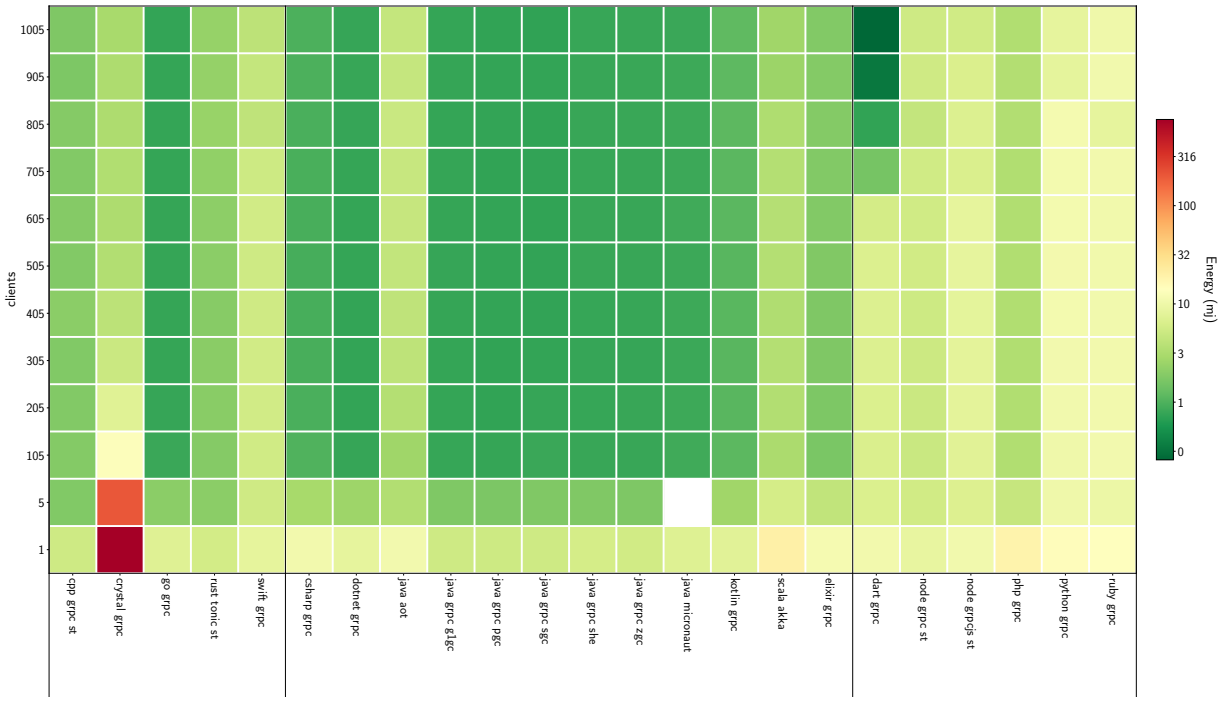


Figure 6.5: The cost of a single request based on the number of clients (mJ)

number of clients does not significantly impact energy consumption. Then, we study how the payload size of the requests impacts the energy consumption of the framework.

RQ 2: How do RPC implementations consume energy concerning 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 category of requests. Therefore, our boundary condition is only the number of requests received by the server. For this experiment, we investigate four observable variables:

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

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

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

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

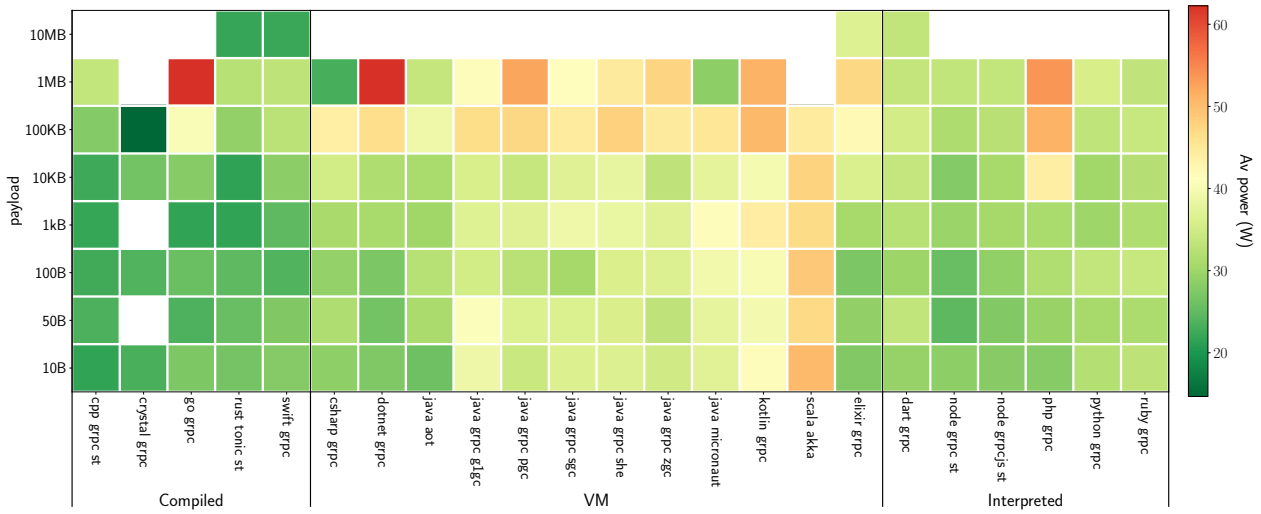


Figure 6.6: Average power consumption based on the request size

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 size of requests exceeds 1MB, most of the servers cannot handle them, and they tend to crash. After 10MB, except Elixir, no server could handle those requests.

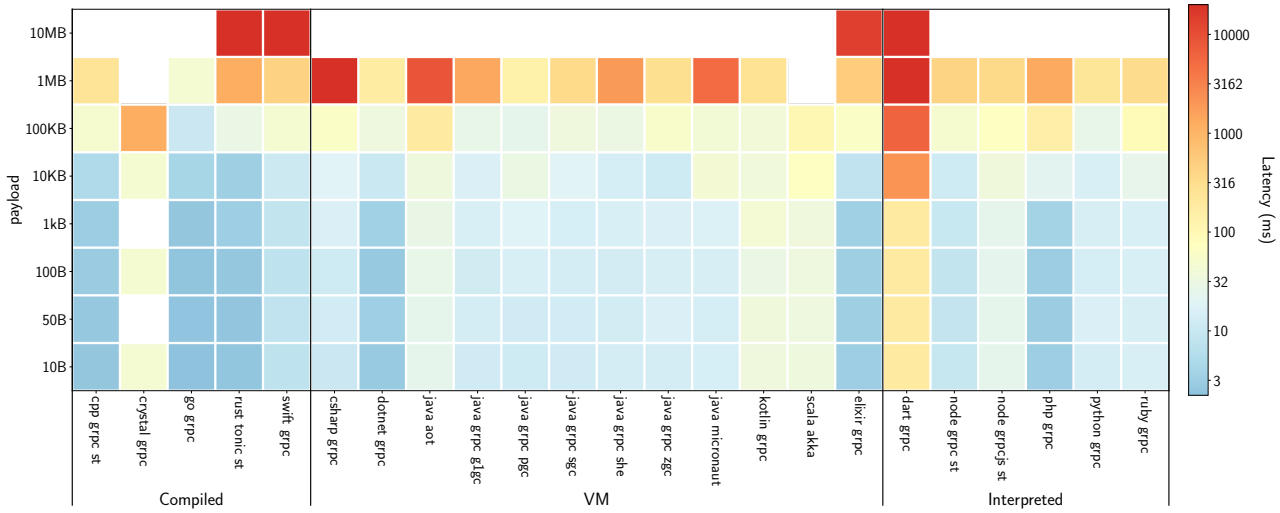


Figure 6.7: Tail latency (99%) based on the request size

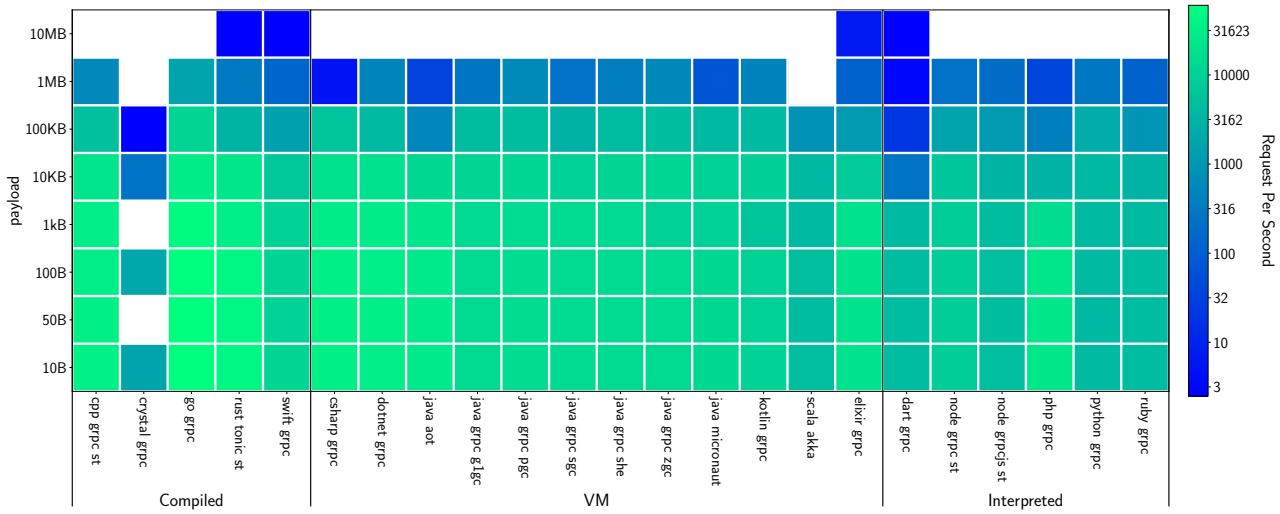


Figure 6.8: Number of requests per second based on the request size

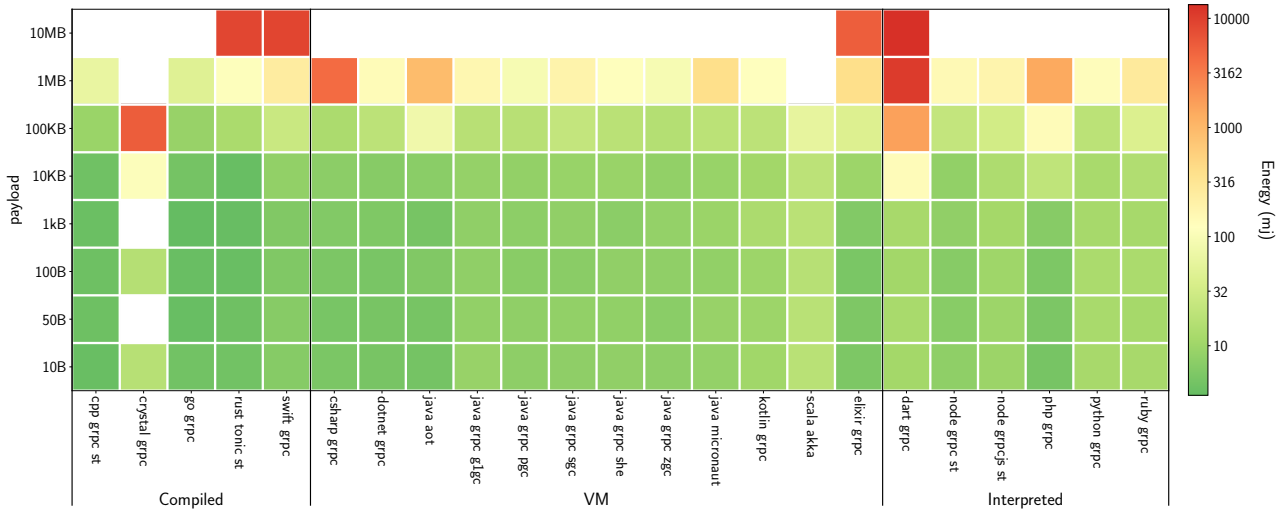


Figure 6.9: Energy consumption based on the request size

Stress-Free Mode

The compiled languages consume fewer resources (average power) in this mode. JVM-based languages tend to consume more energy, especially Scala. However, we do not observe the same behavior regarding efficiency. Unlike the other interpreted programming languages, PHP's performance can be compared to the compiled ones, such as CPP or GO, and even better to others, such as Swift. JVM-based languages tend to have better performance than interpreted ones. Furthermore, OpenJDK has shown more efficiency than GraalVM. Overall, we can have three groups when it comes to 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 source, such as Net core, Java Micronaut, Crystal, and Dart. In this case, the server uses the same resources, sometimes

less, because it takes more time to handle 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 performance, such as Go or.Net. The energy consumption of a single request is affected slightly but still increases.

3. Increase in power while dropping in performance. Despite the increase in 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 performance compared to other compiled languages (Go, CPP).

Broken state mode

Only four of the 25 configurations could parse the 10 MB files, and only one of those could achieve a 76% acceptance rate, which is Elixir. The other 3 had less than a 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 the dynamic codes, such as OpenJDK and Kotlin,

- Size of request exceeded the maximum size when the implementation could not handle requests with a large size, as observed with .Net, Go, .Net core, CPP, PHP, Scala, Nodejs, Ruby, and Python.

6.1.3 Conclusion

This section presents a study on the impact of programming languages on the energy for request handling. To do that, we used an implementation of 25 frameworks based on 17 programming languages. These servers use the official RPC protocol, reducing programmers' bias toward specific languages.

The results show that the number of clients had a binary effect on the servers. With a small number of clients, there was no typical behavior between different categories. On the other hand, when dealing with a more significant number of clients, interpreted languages and compiled ones dropped in terms of performance while keeping the same average power, unlike the Vm based ones that increased their average power while keeping the same performances. Overall the second strategy had a better impact on the single request cost than the first one.

As for the evolution of the query size, we have seen three modes of behavior for each framework, depending on the size of the requests: the stress-free mode, the escalation mode, and the broken state mode. We start with the stress-free mode. The leader class was the compiled languages, while the VM-based ones tended to consume more energy. Finally, with the escalation mode. one could observe three strategies to deal with the increase in request size. The first strategy was to drop in performance while keeping the same power; the second was to increase the average power consumption without impacting the performance; the third was a combination of both previous strategies with an increase in power and a drop in performance. Interestingly the third strategy had a lower cost per request than the first one. Finally, most implementations could not handle requests heavier than 10 MB for the broken mode. Elixir was an exception to this limit despite its low performance compared to other languages

This study shows the absence of a universal language when dealing with RPC requests. Moreover, each implementation had a different way of dealing with scalability. While some chose to increase the average power consumption, others dropped in performance. Both of these strategies led to a higher cost of a single request.

6.2 Investigating Web Application Frameworks

Comparison of Web Frameworks Many studies have been conducted to compare the performance of web frameworks. One can cite [**?**] who compared two of the most famous Java frameworks—Play and Spring—or the work of [**?**] who compared different PHP frameworks using six 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.

6.2.1 Experimental Protocol

This section describes the environment we used during the experiments, covering hardware features, framework experiments, and methodology.

Measurement Context

This experiment aims 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 **??** highlights the number of frameworks used in the experiment per benchmark category. 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, it might be the best choice. Many reasons are behind the observed failures; there was no implementation, or some errors were raised when handling the request.

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 means of the pre-installed power meter,
- database server offers the database that will be used by all the frameworks during the benchmarks.

Table 6.1: 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**

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

The tests have been executed in machines from the cluster chetemi [7] of the grid5000 **??** platform.

TODO: add hardware description

Note It has been proven in the work of **?** that Docker does not impact 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 the 20s for the warmup,
3. measures the average power when the application is in an idle state,
4. using multiple clients, we send the same request concurrently during the 20s,
5. increases 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 its purpose by giving some examples of the expected responses.

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

Test Scenarios We have seven categories of benchmarks:

Idle In this benchmark, we measure the web framework's idle energy consumption; this reflects an application's average energy consumption during periods without connections. For example, a company website beyond working hours or an 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 and 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 will use 512 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). At runtime, an additional fortune cookie message is inserted into the list, 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 untrusted and properly escaped, and the UTF-8 fortune messages must be rendered properly.

Update Queries This benchmark exercises the 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 512. The response is analogous to the multiple-query benchmark.

Plain Text In this benchmark, the framework responds with the most straightforward response: a `"Hello, World"` message rendered as plain text. The response size 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 critical message to the value `"Hello, World!"`. For each of the above scenarios, we consider different stress levels, and Table **??** shows the different levels for each scenario.

Table 6.2: 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, and 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 reach its steady state.

Runtime Measurements

- energy measurement : We use PowerAPI [**?**], 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.

$$E = \int_b^a P(t)dt \simeq \sum_{k=1}^{n} \frac{P(t_k - 1) + P(t_k)}{2} \tag{6.1}$$

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

Due to 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.[8]

Architecture We aim to compare the energy consumption of different web frameworks. For this, we consider the web framework a black box and the response to the six 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 **??** illustrates the architecture of our system.

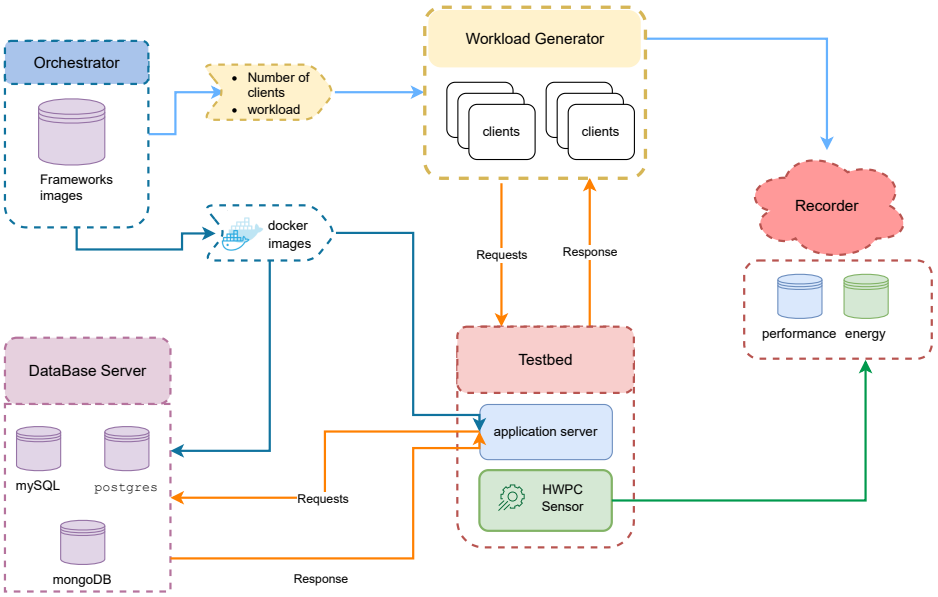


Figure 6.10: Architecture of the experiments.

Bias Analysis We are aware of potential bias analysis regarding estimating 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.

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

Extension

To follow the guidelines that we presented in Chapter **??**, we provide a GitHub repository[9] where one can add extra **candidates** by creating a new project using the option `-new`. Then, interested practitioners must fill out 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.

6.2.2 Results & Findings

Overall we had $8,750$ benchmarks, and all can be found in the online repository.[10] 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 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 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.

[9] https://github.com/chakib-belgaid/FrameworkBenchmarks
[10] 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. We will find a correlation between the metrics to narrow the research space. The Pearson correlation coefficient [**?**] will be used to determine this correlation. Because the Shapiro-Wilk [**?**] test yielded a p-value of 0.0 for all metrics.

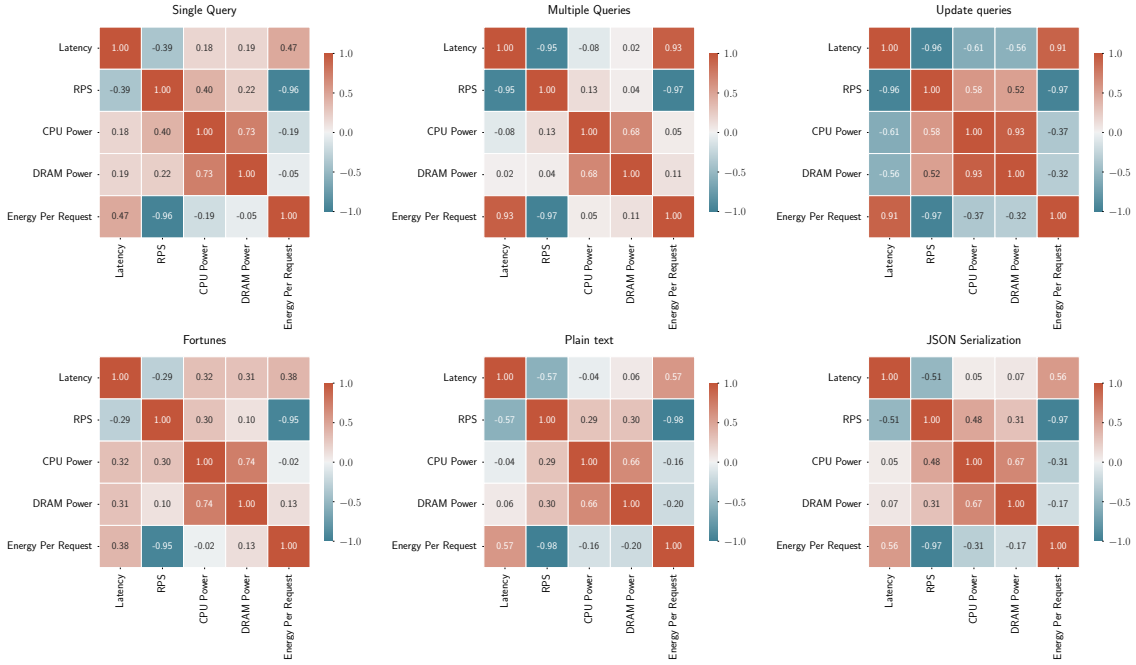


Figure 6.11: Spearman Rank Correlation between different metrics

Figure **??** 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, this study will focus more on CPU energy consumption. For more insights about the DRAM consumption, we refer the reader to our GitHub repository.[11]

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

[11]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 the number of requests per second. The reason behind such an anomaly is the fact that we summarized the data when we had multiple numbers of clients. If we calculate the correlation when we have a fixed number of clients, like in the case of update queries (512 clients), one can notice a strong correlation.

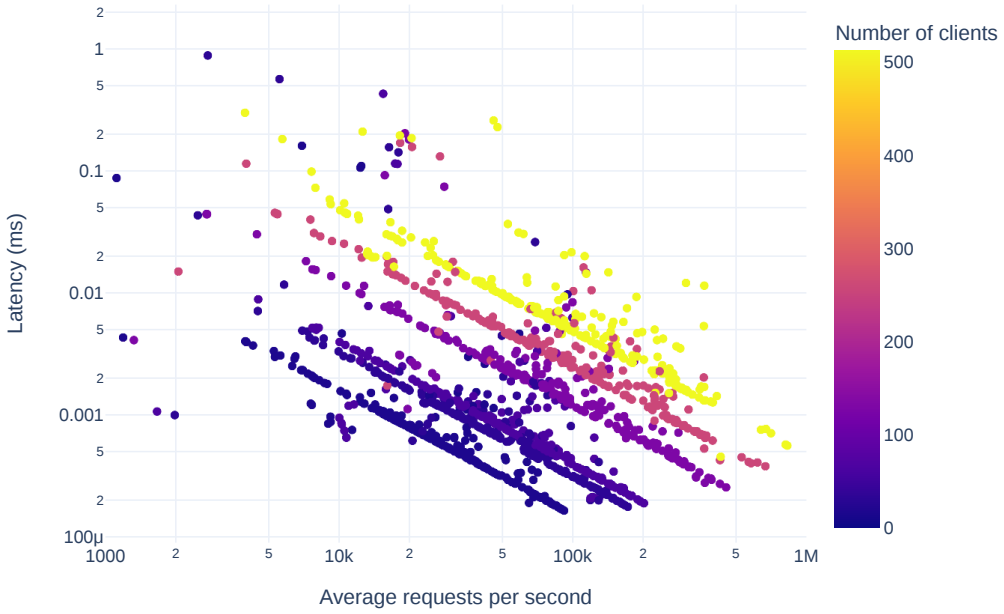


Figure 6.12: Correlation of latency and number of requests per second for a single query

Figure **??** 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 measure how green a framework is.

Scenario-based Analysis

This section will focus on the behavior of all the scenarios' implementations. For visibility purposes, we will group the frameworks not by language but by family so that we will have five families:

- **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 **??** 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 **??** presents, a density plot for each family.

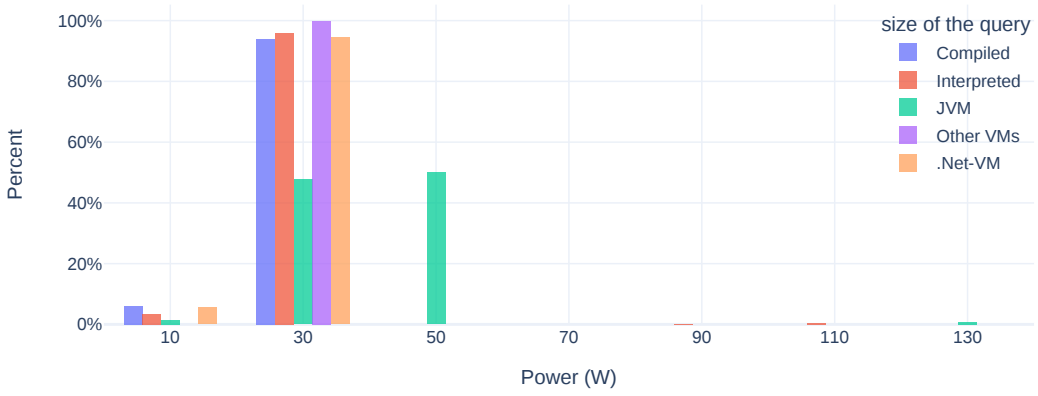


Figure 6.13: Average power consumption for the idle scenario

As one can notice at rest, most families consume between 20 and 40 Watts, and 6% of the compiled language frameworks consume 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 each of the programming languages from Java separately in Figure **??**, we find that Java-based implementations tend to consume around 50 Watts, while Kotlin, Clojure, and Scala consume around 30 Watts.

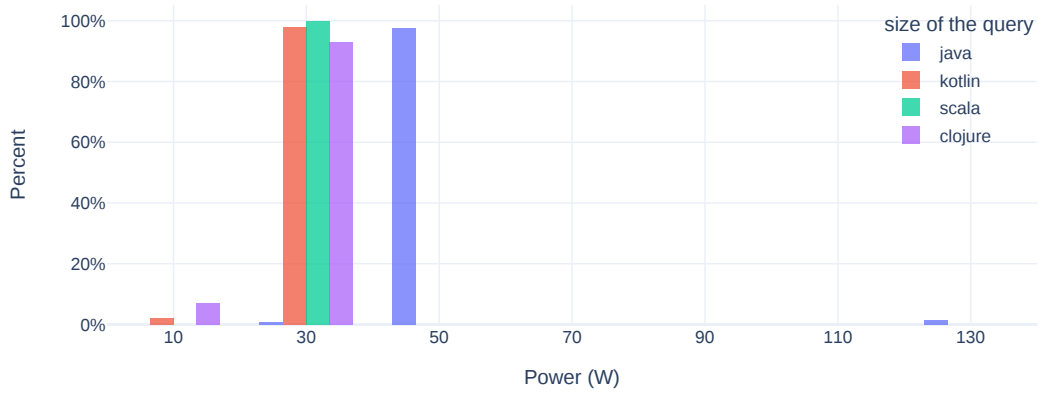


Figure 6.14: Average power consumption for Java-based languages in the idle scenario case

Single query As mentioned before, the purpose of this scenario is to benchmark the framework efficiency to handle a single entry. We will first start with an average power consumption histogram to determine the frameworks' general behavior. Figure **??** reports 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 consume around 70 Watts;

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

3. Finally, the *stress state* beyond 128 concurrent clients: most frameworks have a stable power consumption regardless of the number of clients. This is due to the 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, we include the number of *requests per second* (RPS) as a performance metric of interest. In Figure **??**, each run is represented by a circle, and the size of the circle represents the number of concurrent clients: The smaller the circle the fewer 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 when we increase the number of

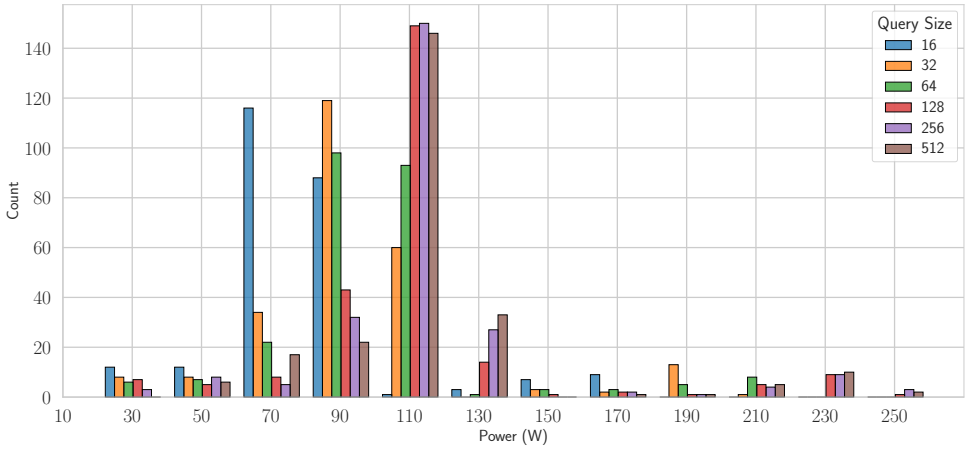


Figure 6.15: Average power consumption for the Single query test

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 performance while keeping low average power, except for PHP, as it has one of the highest RPS with a half million RPS which got beaten only by C++ and Rust.

Multiple queries This scenario benchmarks the framework's efficiency in handling multiple row queries. As mentioned before, this study focuses on a fixed number of concurrent clients while we increase the request size per level. Figure **??** reports on the average power consumption for each level. As one can notice, the query size has no substantial 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 more significant than ten rows. This might be related to the time taken by the database to process the query. Therefore, one can conclude that the query size has more impact on the database than the framework itself.

Table **??** details the average power consumption per level for each database. One can see that for MySQL, there were no changes regardless of the query size, while for Postgres and MongoDB, there is a slight decrease in the average power consumption when the query size is more significant than ten 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 second (aka RPS) as a related performance metric. Figure **??** presents the total RPS per level

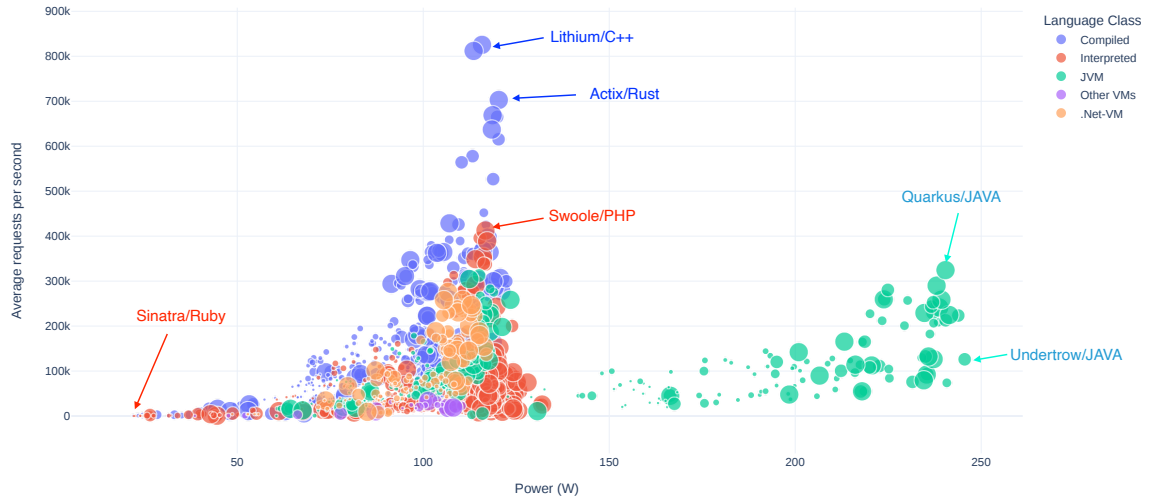


Figure 6.16: Total request vs. average power consumption for the *single query* benchmark (size of circles represents the number of clients)

for each framework on a logarithmic scale. As one can notice, the difference between the best performing framework, aka Lithium,[12] and the worst one, aka hapi-Nginx,[13] is 4,000 times, while the average in power consumption is five times (120 for lithium vs. 25 for hapi). This highlight the importance of the target scale of the application when choosing the framework. Java-based frameworks tend to consume more power compared to other languages, with a

[12]https://matt-42.github.io/lithium/
[13]https://github.com/hapijs/hapi

Table 6.3: 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

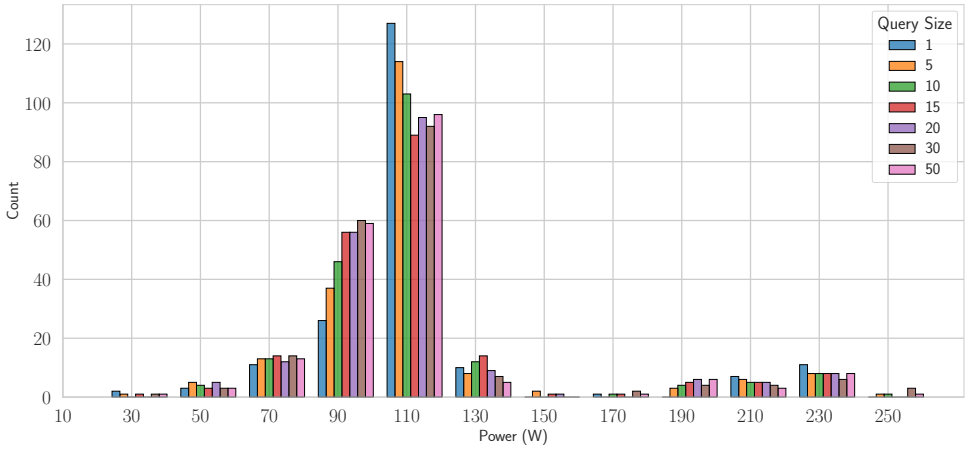


Figure 6.17: Average power consumption for the multiple queries

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 benchmarked the framework's efficiency in handling update queries. As mentioned before, for this study, we will focus on the fixed number of parallel clients while increasing the request size per each level. Figure **??** presents the average power consumption for each level. As one can notice, the query size does not strongly impact the average power consumption. Moreover, the overall average power consumption decreased by 20 Watts.

Figure **??** reports on the number of RPS per level for each framework. Swoole dropped in terms of performance while reducing the average power, unlike compiled languages-based frameworks, such as lithium and actise.net, gained in terms of performance while keeping the same power consumption. Another interesting observation is the linearity between the drop in performance regarding requests with more than ten 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 in other scenarios.

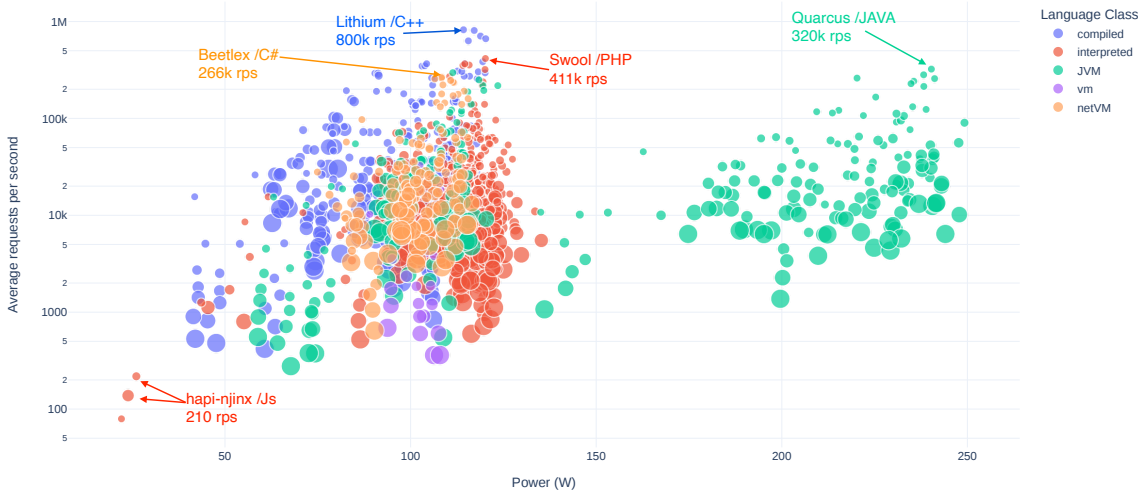


Figure 6.18: Total request vs. average power consumption for the single query benchmark (size of circles represents the number of clients)

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

6.2.3 Summary*TODO : missing*

to conclude, we summarized the results in the following figure

6.3 Conclusion *TODO : missing*

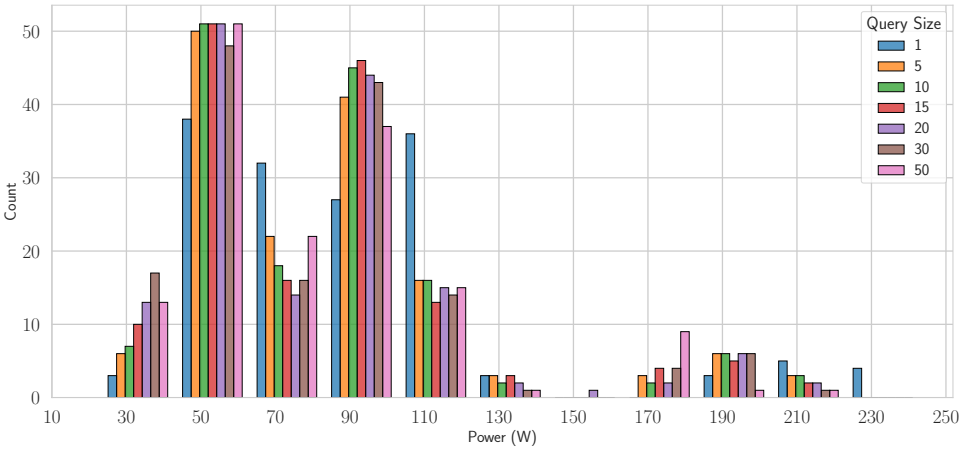


Figure 6.19: Average power consumption for the update scenario

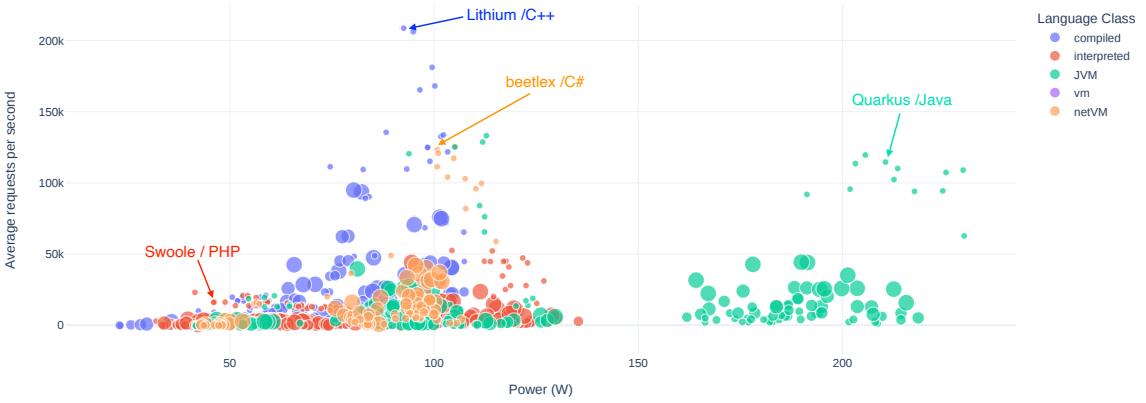


Figure 6.20: Total request vs. average power consumption for the Update benchmark (size of circles represents the number of clients)

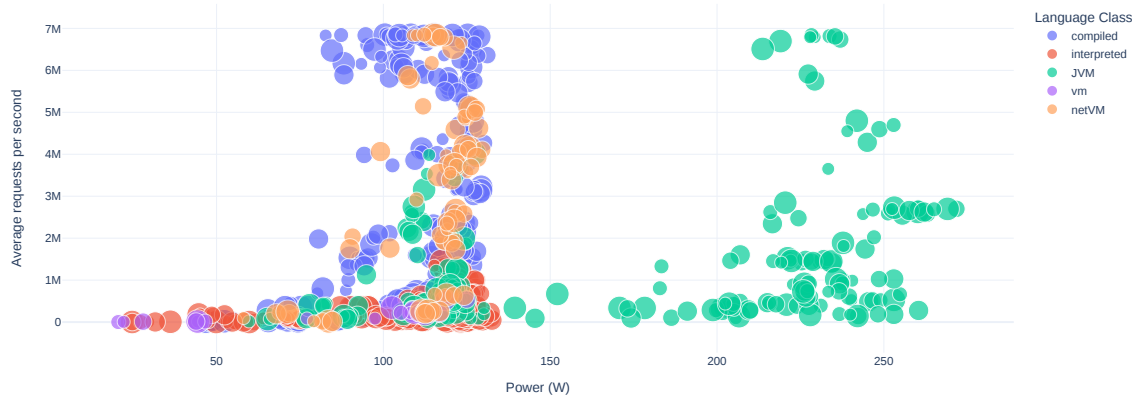


Figure 6.21: Total request vs. average power consumption for plainText benchmark (size of circles represents the number of clients)

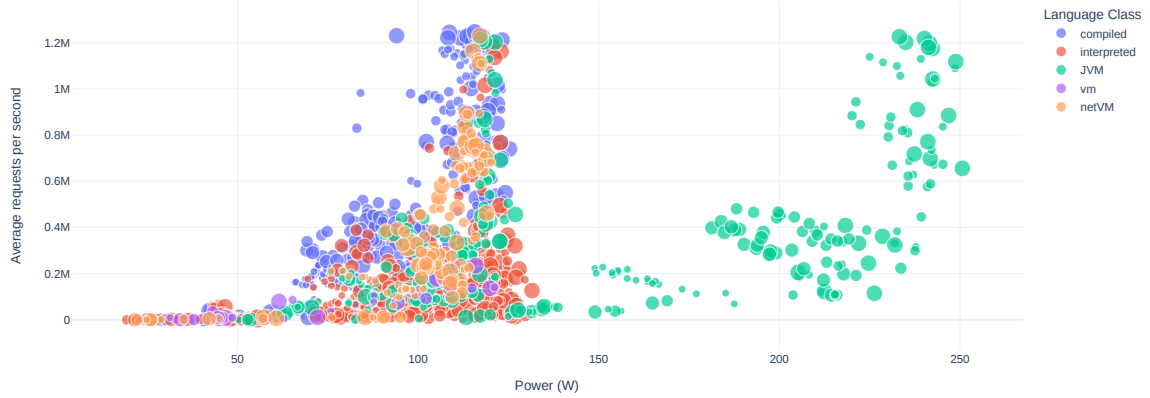


Figure 6.22: total request vs. average power consumption for JSON Serialization test (size of circles represents the number of clients)

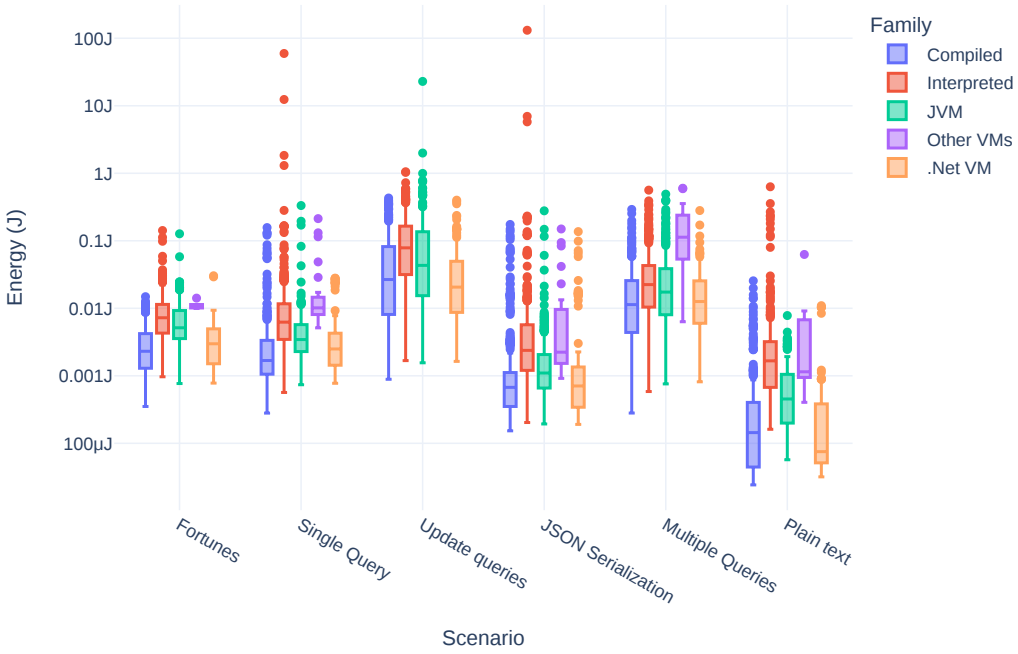


Figure 6.23: Energy consumption per request for each family of programming languages

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. ...