Green Coding

Software energy optimization by understanding the impact of programming languages

Mohammed Chakib Belgaid

Supervisors: Pr. Romain Rouvoy
Pr. Lionel Seinturier

University of Lille

This dissertation is submitted for the degree of
Doctor of Philosophy

EDSPI June 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
June 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

Nowdays, computers are invasing our daily life from work to leasure, from fancy smartphone to an embded peacemaker that regulatres the heartbeat of some people. As human beigns we are known to use tools to enhance our bodies. And thanks to computers we pushed that step even further where now we are using machines to extend our brains. Some even argue that one day they will replace us, and therefore we created our own ending. Well this is the problem for the future generations. For the moment, the main concern of humanity is to keep this planet livebale until we find another alternative.

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

TODO : add the impact of the ict ..etc

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 it'sconsumption. 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 ordrer 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 certains parameters, such as palteform, programming langauge, and/or the design pattern.

The best way to do so is to formulate a theory behind the energy consumption of algorthms 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 algrothim optimization in the field of perfornacne which is agnostic toward the plateform. the energy consumption of the algorithms is dependent on execution environement. Therefore, for the moment we will start by formulating some hypthosis and explore them using empirical analysis.

Our work will be presented through the following chapters :

1. 1.3: Where we discuss the work done on the energy consumption and optimization in software engineering
2. 2.6: It will present a set of guidelines and tools to help practitioners measure the energy consumption of their algorithms.
3. : we will present the impact of programming languages on the energy consumption of the algorithms.
4. : it will discuss the behaviour of python and the possibile ways to tune it in ordrer to reduce the energy consumpion
5. : will present a study on java programming language and the impact of the JVM choice on the energy consumpion
6. : as a prespective we introduce the impact of parallelisme on the enrgy consumption on time agnostic cases

....

1.1 Motivation

....

1.2 Research Contributions

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

1.3 Publications

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

Chapter 2

Literature Review

2.1 benchmarking

this is [**?**]

difference tests [75] [52] [13] list of benchmarks [**?**]:

- Not evaluating potential performance degradation
- Benchmark subsetting without proper justification
- Selective data set hiding deficiencies
- Microbenchmarks representing overall performance
- Throughput degraded by x% => overhead is %
- Creative overhead accounting
- No indication of significance of data
- Incorrect averaging across benchmark scores
- Benchmarking of simplified simulated system
- Inappropriate and misleading benchmarks
- Same dataset for calibration and validation
- No proper baseline
- Only evaluate against yourself

- Unfair benchmarking of competitors
- Not all contributions evaluated
- Only measure runtime overhead
- False positives/negatives not tested
- Elements of solution not tested incrementally
- Missing platform specification
- Missing software versions
- Subbenchmarks not listed
- Relative numbers only

[62]

startup performance vs steady performance [**?**]
simgrid simulation of power consumption of parallel aplication using simgrid [37]
impact of manifacturing process on the energy variation [**?**]
different energy consumption between identical processers [idle and hight load][78]
the temperature is the main raison a high energy variation [**?**]
external factors are impact on the energy variation [60]
the place of the server in the room doesn't affect the energy variation [**?**]
comparaison of three measuring tools and they did exhebit 10% variation each time [40]
low-cost scalable variation-aware algorithm [40]
reducing the energy variation by disabling turbo boost [3]
reducing the energy consumption in parallel systems [17]
[56]

2.2 energy consumption

to be able to reduce the energy cost of running programs we should first be able to estame this energy consumption. many studies have been conducted to estimate a such energy consumption. that varries from static analysis of the source code to enfere it's energy consumption like **?** where they provided a tool to highlight the most energy consuming parts of the code [**?**].the main advantage of this apporach is to be able to estimate the energy consumption of a program without running it. however. unlike the conplexity of programs,

the energy consumption is highly related the execusion environment. Therefor, Static analysis might not represent the real behaviour of the same program when it is run in the production environment. To treat the problem of representativeness, many researchers tend to measure the energy consumption of the programs while they are running them. Hence we will get more accurate results. there is a variety of tools to measure such energy, and they cover a large spectrium of usage depends on how **accurate** and **pricise** those results are needed to be in tone hand, and the price that practinoners are ready to pay for a such accuray/pricision. Below we will present some tools that are well known in the literature and discuss their main advantages / drawbacks for our case.

2.2.1 hardware tools

according to Hackenberg et al. there are four main creterions to evaluate an energy measurement tool [34]

- *spatial granularity: the more specific the target of monitoring we are able to 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 identif the sequence of code that need to be optimized.
- *scalability: this is mainly related to the cost of the tools and the ease of their integration for our system.*
- accuracy:to elimanate extra hazards and get more representative measurement.

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

Nowdays, most of the high performance computing systems (HPC) implements a tool to report the energy consumption of the nodes for the sake of monitoring and administration. Those tools are mainly integrated withing the power suply units (PSU) or the power distributuin units (PDU). then , they provide an interface and a log to follow the history of the energy consumption. Despite their scalability and ease of integration. such tools lack both in spatial and temporal granularity since they monitor the whole energy of the nodes, and most of the times they have a very low sampling frequency. most of those tools are provided directly by the manufacturers. such as *IBM EnergyScale technology [**?**] [**?**] [**?**] or* Dell poweredge

[**?**], MEGware Clusttstafe [**?**] As we said earlier the true purpose of those tools is more monitoring than analysing the energy consumption. WattsUp Pro, is a device that can be installed between the powerer source of the machine and the system under test. it allows a sampling frequency up to 1Hz and has a internal memory to store a wide variety of data, such as the maximum voltage, current.. etc that later can be exported via USB port for personal usage or lined to some graph programs like Logger Pro or LabQuest.The main advantage of this tool is the ability to monitor the energy consumption from a different device which will reduce the risk of interference with the energy consumption of the test [39]

despite his high temporal granularity , wattsup pro lacks in term of spatial granularity since it monitors the energy consumption of the whole system. To have a finer granularity we need to isolate the energy consumption of each component. . For this we will need more intrusive tools such.

PowerMon and its upgradded version powerMon2 [8] are based on an microcontroler chip that can monitor up to 6 channel (8 for powermon2) simultanetly. therefore we are able to monitor the powerconsumption of 4 devices at the same time. the frequency sample of this tool is up to 50Hz with an accuracy of 1.2%. Powermore2 comes with smaller size that can fit within 3.5 inches rack drive.

PowerInsight [48] is another finegrained measurement tool that is based on an ARM bagelbone processor [**?**] which is able to measure up to 30 channel simultanetly with a frequency of 1KHz per channel.

on the other hand GreenMiner [**?**],

powerpack [30] 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 relized by the manufactures such as IBM Power executive [**?**] which allows their customer to monitor the power consumption and thermal behaviour of the of BladeCenter systems in the datacenter.

as we see in The CPU is the part responsible of the most energy consumption of softwares. hence the finner we go to measure this energy consumption the better it is for our work. Fortunately, Intel and later AMD proposed a tool that estimate the power consumption of different part of the CPI based on counter performances. RAPL (RUnning Average Power Limit) [33] [35] is a set of registers that was introduced by intel in their CPU since Sandy bridge generation, and later it was fllowed by AMD since Family 17h Zen. It is capable of monitoring different components of CPU , such as DRAM, Cores , and the integrated GPU for desktop processors.

The advanage of a such apporach is that the absence of any intrusive measurement tools. further more they have a high temporal granurality with a sampling frequency up to 1KHz [**?**].

with similar approach we can find NVIDIA reporting tools such as GPU TESLA [14] and the NVML library [27]

2.2.2 software tools

Now that we got our hands on a precise tools that allows us to monitor

powerapi [**?**]
smartwatts formula [28]
Wattwatcher [49]
joulemeter [45][42]
jrapl [32] [53]
joulinar [41] [65]
jalen [66]
selfwatts [29]
powerjoular [64]

2.2.3 hybrid tools

2.3 programming languages and performances

2.3.1 energy

energy comparaison of programming languages in the game benchmark by debian [**?**]

toward green ranking [21]
java vs kotlin in web performances [12]
rosetta code [63] [57]
network energy comparaison [4]
Aequitas [73]
vm placement [59]
Mishra et al.
static cost of Vms [46]
carbon footprint of training neural network [**?**]
SPELL, the energy leaks detector tool [**?**]
impact of energy profiling on software [42]

impact of maintability on software energy evolution [15]
defintion [79]
comparaison [16]
emprical study [22]
impact of website implemenation on energy consumption [**?**] [**?**]
PHP [**?**] [**?**]
simulation of web cache [**?**]
java spring analysis [**?**]
performance [58]
cross compiler benchmarking [81]

2.3.2 python

webframeworks on python [**?**]
hope library [**?**]
bytecode effect [**?**]
numba [**?**]
intel python [**?**]
python 2 vs 3 [**?**]
python runtime performances [**?**] [61]
static vs dynamic languages [**?**]
concurent benchmark framework [69]

2.3.3 jvm

java vs python [**?**]
hotspot vs J9 [**?**] - same but for big querries [**?**]
constant overheat of the jvm [47]
infer the energy cost based on the byte code instructions [54]
java collection energy footprint [71] [**?**]
java classes energy footprint [**?**]
framework to reduce java collections [**?**]
energy consumption of java dev tools [7]
Microbenchmarks on jvm [**?**] [7]
android automatic refactoring to reduce energy consumption [6] [**?**]
java vs native c in android [20]

2.3.4 benchmarking

NAS [**?**] statistics [**?**] benchmark crimes [**?**] microservices [**?**] impact of webservers on web applications energy [55]

2.4 Reproducibility in benchmarking

2.4.1 Virtual machines

To resolve this problem, practitioners tend to use Virtualisation. Using virtual machines aka VM gives reaserchers the freedom to choose their own tools, software and operating system that they are the most confortable with without paying the price to change the actual working environment, which will give them eventually more controle over the dependencies and the execution environement. Moreover, using a vm will solve the *replication crisis* thanks to the virtual images, even the most complex architecture can be reproduced easily by just instantiating a copy of the image. Since the virtual machines are agnostic to the host architecture, reaserchers won't have to worry about where and how their experiments are replicated because they have already setup the execution environement. Another advantage to the virtual machines is the snapshot mechanism, it allows reaserchers to create backups and revert some changes with simple clicks. Last but not least,thanks to the isolation, virtual machines push the reproducibility further by allowing the future usages to see all the variables -controled and uncontroled- and do other analysis without dealing with any dependencies. In his paper [**?**] bill howe lists the advantages of using virtual machines in reaserchers experements including the economical impact and cultural limitation to a such approach.

which allow them to have control over the ressouces, the dependencies and the execution environment. Moreover, thanks to the snapshots, deploying a software is easily done by instantiating a copy of that image. However, this choice comes with a certain cost. Because the intervention of the hypervisor,the software will use two kernels , the virtual machine one and the host machine one, which will provide a noticiable overhead, and will impact the performances of the tests.Therefore, we can't use virtual machines for exepements that are related to performance. Another limiation with the virtual machine is the the isolation. It is true that this feature will prevent the experience environement with any undesirable interference from the outside world. but sometimes this contact is needed, especially when the experiment is dependent to an external part, such as sensors. In energicial tests we tend to use hardware powermeters which will make it difficult to use the virtual machines in this case .

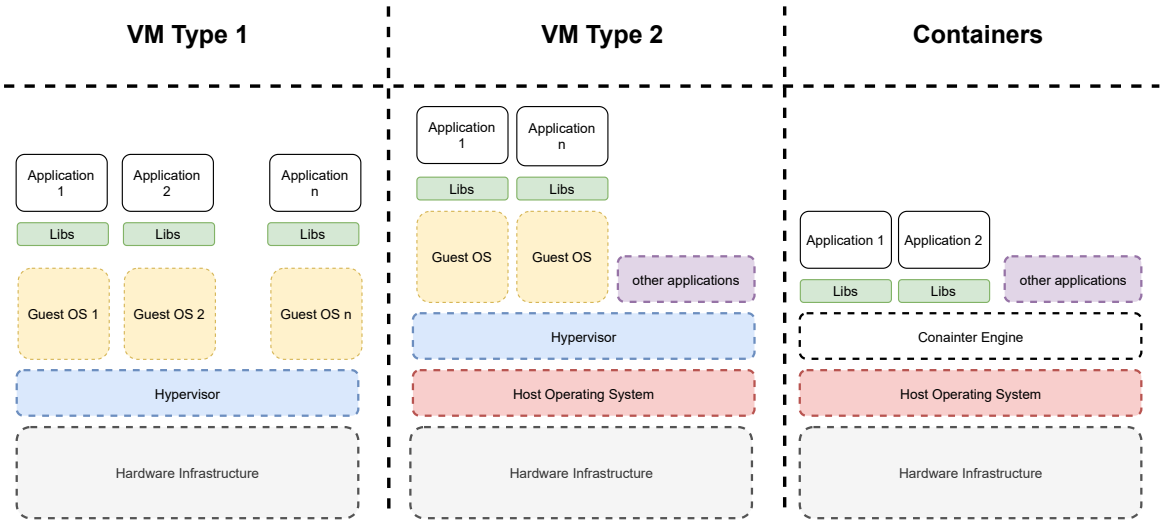


Figure 2.1: Different Methods of Virtualzation

2.4.2 Containers

another solution would be using something that allows us to have the isolation from the host os and the ease of replication that virtual machines offer, and the direct interaction with the hardware that the classical method give. contarization offers a such advantages while keeping the isolation and the ease of replication for application.

Figure 2.1 explains the differents in architecture between the clasic types2 of Virtualisation and Containers.

- Type 1: runs directly on the hardware, it is mainly used by the cloud providers where there is no main OS, but just virtaul machines, we can site for this the open-srouce XEN and VMware ESX

- Type 2: runs over the hostmachine Operating System, mostly used for personal comptuers, VMware server and virtualBox are famous examples of this type, most of the reaserchers expermentation are run witht this type, howerver due to the 2 Operatings syestms the applications tend to be more slower

- containers : Instead of its own kernel, containers used the hots kernel to run their Os, which makes them ligher, quickers and use the full pontial use of the hardware. For this we can cite *Docker*, *Linux LXCLXD* [**?**]

2.4.3 Docker vs Virtual Machine

Depsite that Type 1 is more performant than type 2, the second one is the most used in reaserch, since most researchers conducts their experements in their own machine. In the other hand, docker is the most famous thechonology for for containers. In ower case we are more prone to docker for two reasons.

1. we need a litetweight orchestrator to not affect the energy consumption of ower tests. As prior work mentioned [cite Morabito (2015) and van Kessel et al. (2016)]

2. since we are using the hardware itself to measure the energy consumption, we are required to interract with the host OS itself.

Special notice to virtualwatts. A framework that allows us to retireve the energy consumption of a virtual machine.

2.4.4 Docker and energy

Now that we have chosen to go with the containers technorolgy to encapsulate our tests. What would be the impact of this solution on the energy consumption of our tests.

Based on the studies of [**?**], who analysed the impact of adding the docker layer on the energy consumption. In their experement. Eddie Antonio et al run multiple benchmarks with and without docker. and compared their energy consumption and execution time. The first step was to see the impact of docker deamon while there is no work. to see the impact of the orchestrator alone. Later they had the experiment with the following benchmarks

- wordpress
- reddis
- postfresSQL

The following figures represents the energy consumption of the system while it is idle. As we can see in figure 2.2 Docker braught arround 1000joules overhead. In the other hand as we can see in figure 2.3. docker increased the execution time of the benchmark by 50 seconds which caused an increase in energy since they are highly correlated. The authors also highlated the fact that this increase of energy consumption is due to the docker deamon and not the fact that the application is in a container. Moreover they estimted the price of this extra energy and it was less than 0.15$ in the worst case. Which is non significant compared to the advantages that docker bring for isolation and reproducibility.

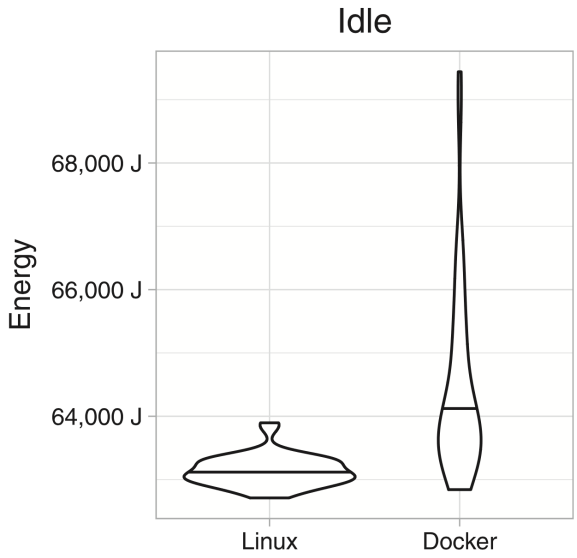


Figure 2.2: energy consumption of Idle system with and without docker [**?**]

if we recap this study in one sentence,it would be the following one. The dockerised softwares tend to consume more energy, because mainly they take more time to be executed. The average power consumption is higher with only **2Watts** and it is due to the docker deamon.This overhead can be up to 5% for IO intensive application, but it is mearly noticiable when it comes to CPU or DRMA intesive works

2.5 Energy Variation

2.5.1 Studying Hardware Factors

This variation has often been related to the manufacturing process [18], but has also been a subject of many studies, considering several aspects that could impact and vary the energy consumption across executions and on different chips. On the one hand, the correlation between the processor temperature and the energy consumption was one of the most explored paths. Kistowski *et al.* showed in [**?**] that identical processors can exhibit significant energy consumption variation with no close correlation with the processor temperature and performance. On the other hand, the authors of [80] 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.

TODO : add the corelation value

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

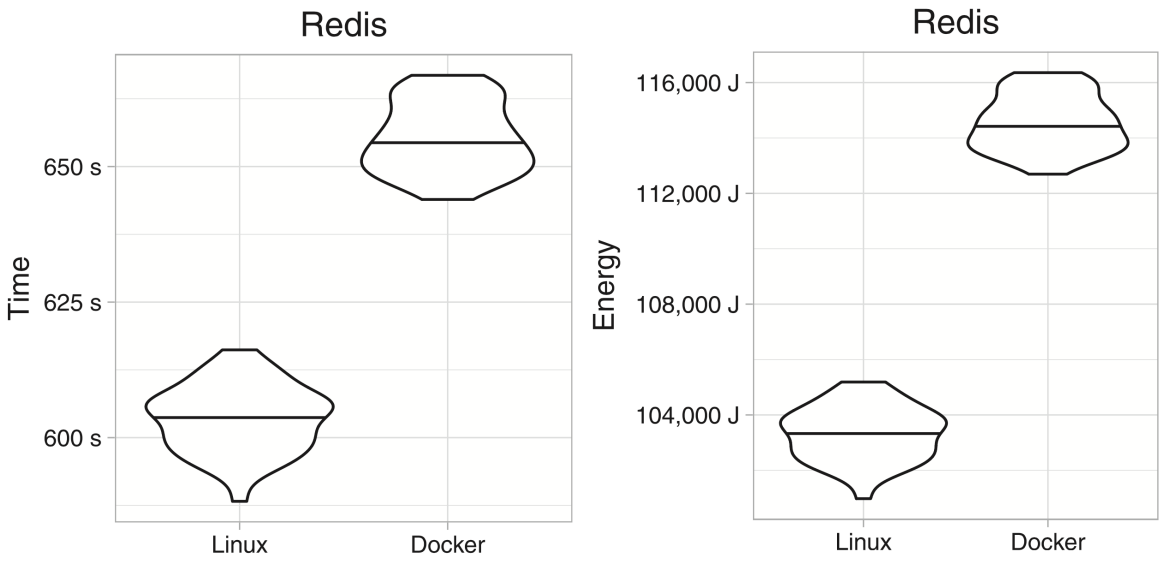


Figure 2.3: execution time and energy consumption of Redis with and without docker [**?**]

The ambient temperature was also discussed in many papers as a candidate factor for the energy variation of a processor. In [77], the authors claimed that energy consumption may vary due to fluctuations caused by the external environment. These fluctuations may alter the processor temperature and its energy consumption. However, the temperature inside a data center does not show major 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 [80], where the rack placement and power supply introduced a maximum of 2.8 % variation in the observed energy consumption.

Beyond hardware components, the accuracy of power meters has also been questioned. Inadomi *et al.* [**?**] used three different power measurement tools: RAPL, Power Insight[1] and BGQ EMON. All of the three tools recorded the same 10 % of energy variation, that was supposedly related to the manufacturing process.

2.5.2 Mitigating Energy Variations.

Acknowledging the energy variation problem on processors, some 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

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

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 out a way to reduce the energy variation on Ivy Bridge and Sandy Bridge processors, by disabling the Turbo Boost feature to stabilize the execution time over a set of processors. They also proposed some guidelines to reduce this variation by replacing the old—slower—chips, by load balancing the workload on the CPU cores and leaving one core idle. They claimed that the variation between the processor cores is insignificant. In [**?**], the researchers showed how a parallel system can be used to deal with the energy variation by compensating 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.

2.6 Conclusion

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

1. execute the code
2. measure the program
3. infere the guidelines

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

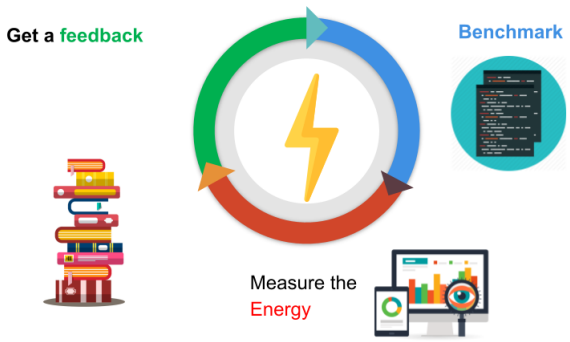


Figure 2.4: the spiral methode of energy optimization

to compare the energy consumption of different programming languages. We will extend the work of **?** to a closer distance to production environment by comparing a set of usecases. starting by GRPC framework, and a set of Web Framewroks. Finally we will discuss the impact of the execution environment on the energy consumption of two of the most famous programming languages JAVA and Python. and present how can tunning the Virtual Machine can reduce the energy consumption.

Chapter 3

Benchmarking Protocol for measuring Energy Consumption of stoftwares

3.1 Introduction

To optimize the software energy consumption, a reproducible setup environment requires to be designed.

This chapter will be dedicated to provide a set of guidelines to provide a *reproducible, accurate and* representative tests. Each aspect will be detailed in the sections below.

3.2 Threats and Challenges

A successful benchmark has three criterions to fullfil. First, it has to be *reproducible* for others to replicate. Second, the results should be *accurate*, which means each time we rerun the benchmark we are expecting the same results. Finally, it should *represent* the real world. In other words, the conclusions brought from the experiment should be valid outside the experimentation as well. In my case, the real world is the production environment. Therefore, our experiments should reflect what is happening in the production environments. In this section, I will deeper dig in each aspect, present what has been done by others and my contributions, and finally I will propose some perspectives, to improve such experiments.

3.2.1 Reproducibility

One of the most difficult challenges faced by researchers is the reproducibility of their benchmarks. Actually, many results fail to be reproduced,[1] which led to a *replication crisis*. As this crisis hit most of the empirical studies, most of the reviews now includes reproducibility as one of the minimal standard for judging scientific merit.[70] One of the criterions supporting reproducibility is the publication of the dataset and the algorithms run on the raw data to conclude the results. There is even some disagreement about what the terms "reproducibility" or "replicability" by themselves mean [31]. According to [24], *replicability* extends *reproducibility* with the ability to collect a new raw dataset comparable to the original one by re-executing the experiment under similar condition, instead of just the ability to get the same results by running the statistical analyses on the original data set.

In the area covered by this PhD thesis, reproducibility might be achieved by ensuring the same execution settings of physical nodes, virtual machines, clusters or cloud environments. However, when it comes to measuring the energy consumption of a system, applying acknowledged guidelines and carefully repeating the same benchmark can nonetheless lead to different energy footprints not only among homogeneous nodes, but even within a single node.

One major problem that hinders the reproducibility of the empirical benchmarks is the interaction with the external environment, either as concurrency or dependencies. Therefore, researchers cannot observe the same results, unless they duplicate the same environment.

3.2.2 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 ny case, this means the ability to run the benchmark multiple times with a low variation.

Recently, the research community has been investigating typical "crimes" in systems benchmarking and established guidelines for conducting robust and reproducible evaluations [**?**].

In theory, using identical CPU, same memory configuration, 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 with in the same configuration are not sufficient to reproduce the the same energy measurements, not only between identical

[1] 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.

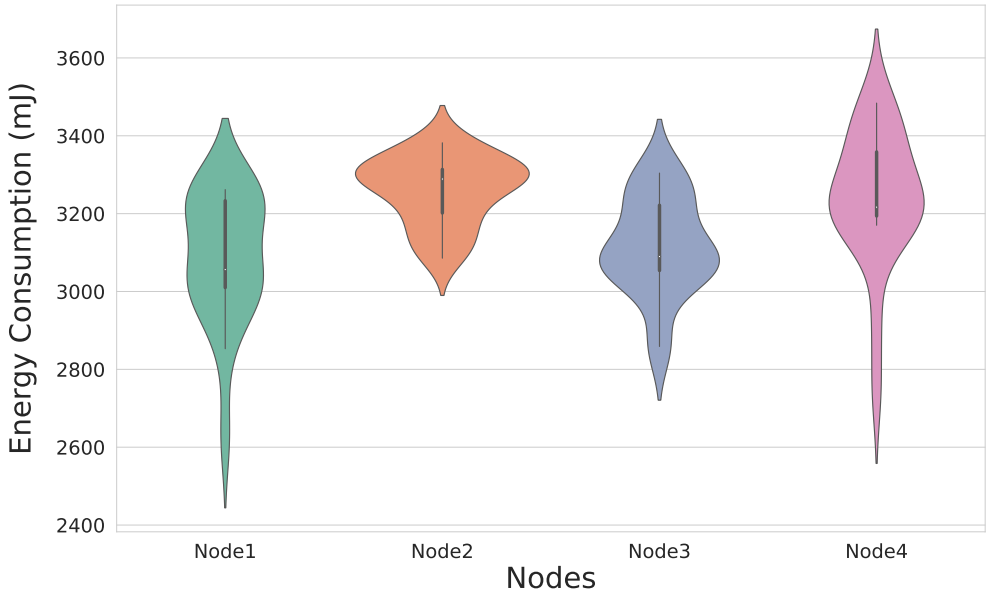


Figure 3.1: CPU energy variation for the benchmark CG

machines, but even within the same machine. This difference—also called *energy variation* (EV)—has become a serious threat to the accuracy of experimental evaluations.

Figure 3.1 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 3.1) at 50 % workload. We can observe a large variation of the energy consumption, not only among homogeneous nodes, but also at the scale of a single node, reaching up to 25 % in this example.

Some researchers started investigating the hardware impact of the energy variation of power consumption. As an example, one can cite [11, 76] who reported that the main cause of the variation of the power consumption between different machines is due to the **CMOS** manufacturing process of transistors in a chip. [38] described this variation as a set of parameters, such as CPU Frequency and the thermal effect.

3.2.3 Representativeness

As obvious as it seems, the reason of executing benchmarks is to validate ideas so we can use them in the real life. However, this is means that those benchmarks have to represent reality somehow. Basically, when we aim to benchmark something, we create a mock up

version of the situation that we want it to work. First, we can talk about the benchmarking and their selection, then we will talk about the stress benchmark for some applications, and finally we will bring this representativeness in our case and how can we get closer to the energy consumption behavior of a software between the benchmark environnement and the production one.

As Stephen M. Blackburn *et al.* cited in their paper "evaluate collaboratory" [9], one of the major pitfalls of the measurement contexts is the inconsistency, which can be translated here by the fact that the production context is not the same as the benchmarking one.

Another difficult part for practitioners is to generalize the claims they reached beyond the lab conditions. Are they appropriate? Are they consistent and are they reproducible? To answer those questions, the community agreed on some wellknown benchmarks to represent a specific concern of the production world. One can site as an example the Dacapo and Renaissance benchmark suites for Java applications, or the CLBG benchmark suite for comparing programming languages. Although they do not cover all the cases, the community agrees on their relevance and representativeness.

In addition to those benchmarks, a new category of benchmarking technique emerged to simulate the worst case of the production environments, inlcuding performance tests, which are benchmarks meant to evaluate the behavior of a software under stress situations. We can cite as an example the Gatling for web application and stress-ng for hardwares.

3.2.4 docker and accuracy

And now Since the stat of the art has aggreed on the impact of docker on the energy consumption,Let's discuss it's impact on the accuracy. In other words
RQ : does Docker affect the energy variation of the exepements ?

To Answer this question we have conducted a preliminary experiment by running the same benchmarks LU, CG and EP in a Docker container and a flat binary format on 3 nodes of the cluster Dahu to assess if Docker induces an additional variation. Figure 3.2 reports that this is not the case, as the energy consumption variation does not get noticeably affected by Docker while running a same compiled version of the benchmarks at 5 %, 50 % and 100 % workloads. In fact, while Docker increases the energy consumption due to the extra layer it implements [25], 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.

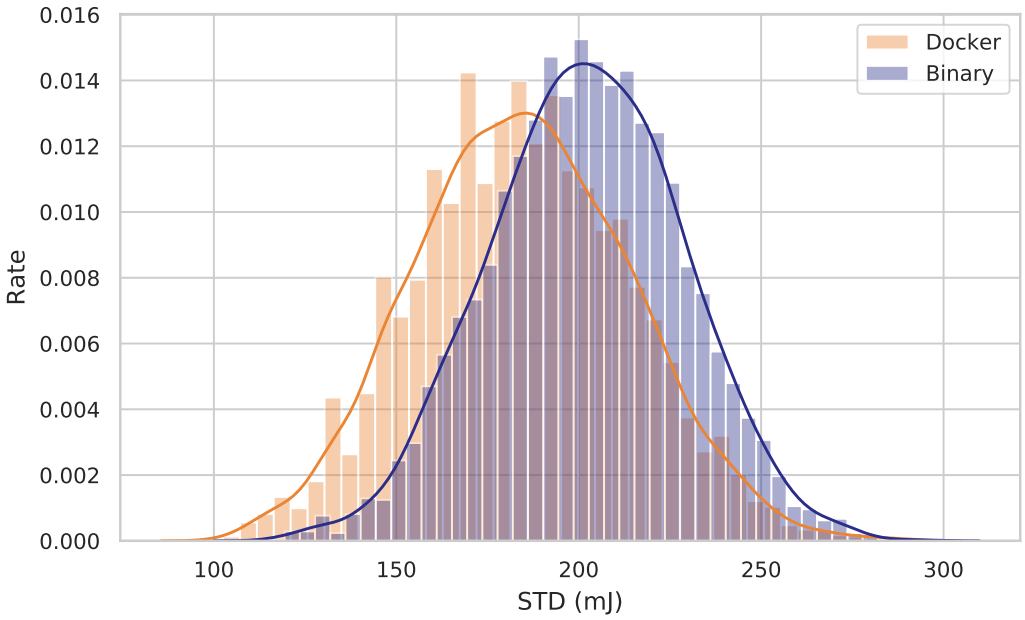


Figure 3.2: Comparing the variation of binary and Docker versions of aggregated LU, CG and EP benchmarks

3.3 Taming the energy variation for the benchmarking protocol

3.3.1 Objective

While the previous part aims to create a large umbrella that covers all empirical benchmarks related to computer science, we rather aim to narrow our study to energy benchmarking only. In other words, we will focus on pitfalls that hinders the energy claims of software. We are clearly aware of the impact of hardware on energy variations. However, we believe that there is still a room to control this energy variation for practitioners by using only parameters that can be tuned. To do so, we have inducted a set of empirical experiments using the guidelines provided by state of the art, to determine which controllable factors can reduce the variation of the energy consumption of benchmarks.

In this part we will try to answer 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 is the impact of the operating system on the energy variation? and finally

RQ 4: Does the choice of the processor matter to mitigate the energy variation?

3.3.2 Experimental Setup

This section describes our detailed experimental environment, covering the clusters configurationand the benchmarks we used justifying our experimental methodology.

Measurement Context

There are three main contexts.

- Different machines with different configuration
- Different machines with the same configuration
- Same machine

To satisfy those requirements we have used the plateform Grid5000 (G5K) [5**?**],a large-scale and flexible testbed for experiment-driven research distributed accross all france. Grid5000 offers miultiple clusters composed with 4 upto 124 identical machine with different configurations for each cluster. For our experement we have considered 4 clusters. Our main creterions was the CPU Configuration. the table below 3.1, presents a description of the 4 clusters

Table 3.1: Description of clusters included in the study

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

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

As an example, Figure 3.3 illustrates a detailed topology of a node belonging to the cluster Dahu.

Workload

Our choice for the benchmarks was based on two creterions.

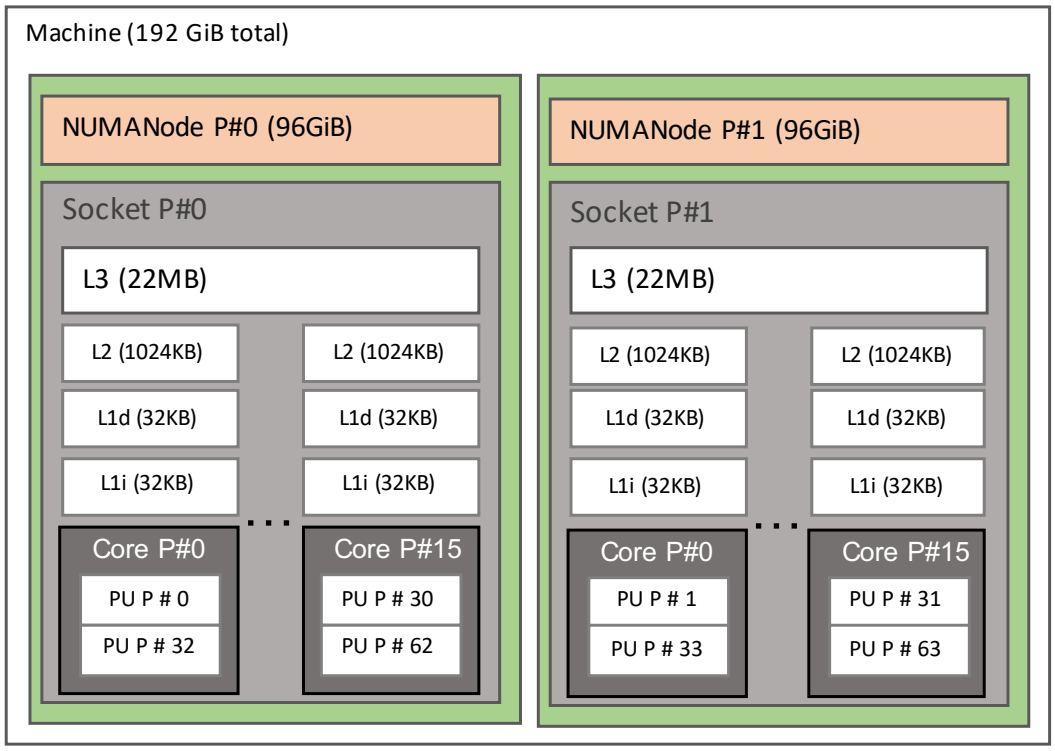


Figure 3.3: Topology of the nodes of the cluster Dahu

First, **scalability** : We wanted to gather the highest possible amount insights (regarding time spent for the experiment, therefore we wanted some benchmarks who can scale according to the number of PUs and can fulfill different scenarios. Second creterion is **representativeness** : as menstionned in the challenges, a workload has to be representative otherwise the experiment would be inconsistent [9]. To satisfy those creterions we went through state of the art and looked for the most used benchmarks for testing the performance of the hardware, and then we selected the scallable ones. Our candidate is emphNAS 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. Further more 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].

[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

Metrics & Measurement Tools

Our metric for the accuracy of the test is the Standard deviation aka **STD** of the energy consumption. Therefore wether the tests consumes more or less energy is out of our scope. To study this variation we need first a tool to measure the energy consumption. For this we used POWERAPI [19], which is a power monitoring toolkit that is based on *Intel Running Average Power Limit* (RAPL) [44]. The advantage of PowerAPI is that it reports the Energy consumption of CPU and DRAM at a socket level.

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

Every experiment is conducted on 100 iterations, on multiple nodes and using the 3 NPB benchmarks we mentioned, with a warmup phase of 10 iterations for each experiment. In most cases, we were seeking to evaluate the *STandard Deviation* (STD), which is the most representative factor of the energy variation. We tried to be very careful, while running our experiments, not to fall in the most common benchmarking "crimes" [**?**]. As we study the STD difference of measurements we observed from empirical experiments, we use the bootstrap method [26] to randomly build multiple subsets of data from the original dataset, and we draw the STD density of those sets, as illustrated in Figure 3.2. Given the space constraints, this paper reports on aggregated results for nodes, benchmarks and workloads, but the raw data we collected remains available through the public repository we published.[7] We believe this can help to achieve better and more reliable comparisons. We mainly consider 3 different workloads in our experiments: single process, 50 %, and 100 %, to cover the low, medium and high CPU usage when analyzing the studied parameters effect, respectively. These workloads reflect the ratio of used PU count to the total available PU.

[6]https://github.com/grid5000/environments-recipes/blob/master/debian9-x64-min.yaml
[7]https://github.com/anonymous-data/Energy-Variation

3.3.3 Analysis

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

RQ 1: Benchmarking Protocol

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

In this first experiment, we investigate if changing the testing protocol affects the energy variation. To achieve this, we considered 3 execution modes: In the "normal" mode, we iteratively run the benchmark 100 times without any extra command, while the "sleep" mode suspends the execution script for 60 seconds between iterations. Finally, the "reboot" mode automatically reboots the machine after each iteration. The difference between the normal and sleep modes intends to highlight that the CPU needs some rest before starting another iteration, especially for an intense workload. Putting the CPU into sleep for several seconds could give it some time to reach a lower frequency state or/and reduce its temperature, which could have an impact on the energy variation. The reboot mode, on the other hand, is the most straightforward way to reset the machine state after every iteration. It could also be beneficial to reset the CPU frequency and temperature, the stored data, the cache or the CPU registries. However, the reboot task takes a considerable amount of time, so rebooting the node after every single operation is not the fastest nor the most eco-friendly solution, but it deserves to be checked to investigate if it effectively enhances the overall energy variation or not.

Figure 3.4 reports on 300 aggregated executions of the benchmarks LU, CG and EP, on 4 machines of the cluster Dahu (cf. Table 3.1) for different workloads. We note that the results have been executed with different datasets sizes (B, C and D for single process, 50 % and 100 % respectively) to remedy to the brief execution times at high workloads for small datasets. This justifies the scale differences of reported energy consumptions between the 3 modes in Figure 3.4. As one can observe, picking one of these strategies does not have a strong impact on the energy variation for most workloads. In fact, all the strategies seem to exhibit the same variation with all the workloads we considered—*i.e.*, the STD is tightly close between the three modes. The only exception is the reboot mode at 100 % load, where

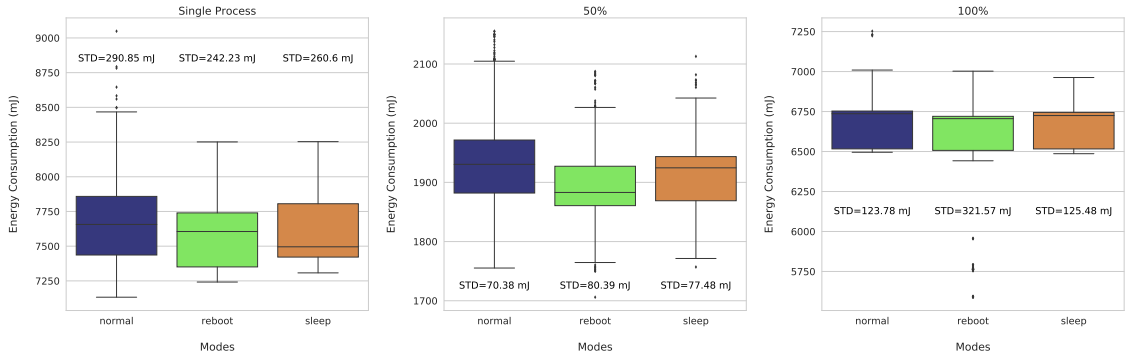


Figure 3.4: Energy variation with the normal, sleep and reboot modes

the STD is 150 % times worst, due to an important amount of outliers. This goes against our expectation, even when setting a warm-up time after reboot to stabilize the OS.

In Figure 3.5, we study the standard deviation of the three modes by constituting 5,000 random 30-iterations sets from the previous executions set and we compute the STD in each case, considering mainly the 100 % workload as the STD was 150 % higher for the reboot mode with that load. We can observe that the considerable amount of outliers in the reboot mode is not negligible, as the STD density is clearly higher than the two other modes. This makes the reboot mode as the less appropriate for the energy variation at high workloads.

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

RQ 2: Processor Features

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

We intend to investigate how much the energy consumption varies when disabling the C-states (thus, keeping the CPU in the C0 state) and at which workload. Figure 3.6 depicts the results of the experiments we executed on three nodes of the cluster Dahu. On each node, we ran the same set of benchmarks with two modes: C-states on, which is the default mode, and C-states off. Each iteration includes 100 executions of the same benchmark at a given workload, with three workload levels. We note that our results have been confirmed with the benchmarks LU, CG and EP.

We can clearly see the effect that has the C-states off mode when running a single-process application/benchmark. The energy consumption varies 5 times less than the default mode.

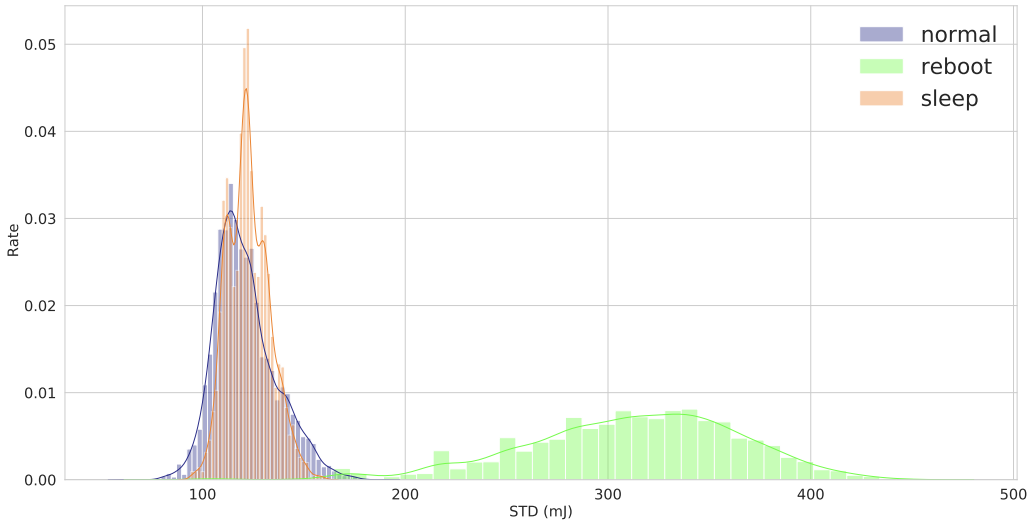


Figure 3.5: STD analysis of the normal, sleep and reboot modes

In this case, only one CPU core is used among 2×16 physical cores. The other cores are switched to a low consumption state when C-states are on, the switching operation causes an important energy consumption difference between the cores, and could be affected by other activities, such as the kernel activity, causing a notable energy consumption variation. On the other hand, switching off the C-states would keep all the cores—even the unused ones—at a high frequency usage. This highly reduces the variation, but causes up to 50 % of extra energy consumption in this test ($Mean_{C-states-off} = 11,665mJ$,$Mean_{C-states-on} = 7,641mJ$).

At a 100 % workload, disabling the C-states seems to have no effect on the total energy consumption nor its variation. In fact, all the cores are used at 100 % and the C-states module would have no effect, as the cores are not idle. The same reason would apply for the 50 % load, as the hyper-threading is active on all cores, thus causing the usage of most of them. For single process workloads, disabling the C-states causes the process to consume 50 % more energy as reported in Figure 3.6, but reduces the variation by 5 times compared to the C-states on mode. This leads to mainly two questions: Can a process pinning method reduce/increase the energy variation? And, how does the energy consumption variation evolve at different PU usage level?

Cores Pinning To answer the first question, we repeated the previous test at 50 % workload. In this experiment, we considered three cores usage strategies, the first one (S1) would pin the processes on all the PU of one of the two sockets (including hyper-threads), so it will be used

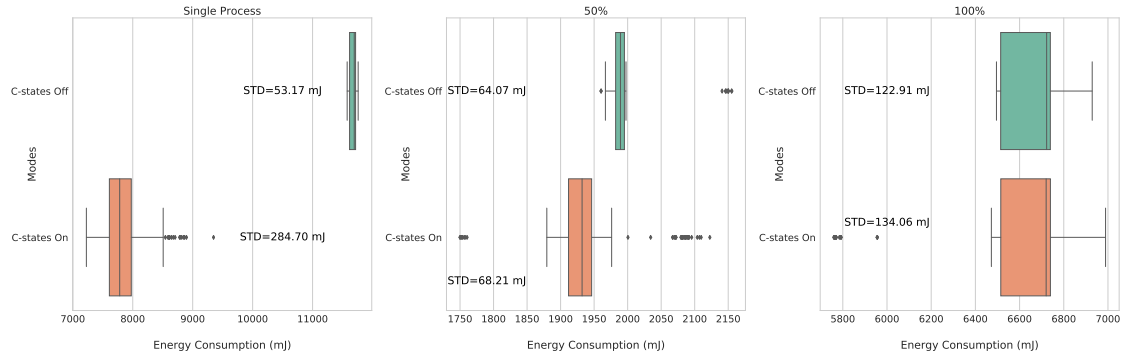


Figure 3.6: Energy variation when disabling the C-states

at 100 %, and leave the other CPU idle. The second strategy (S2) splits the workload on the two sockets so each CPU will handle 50 % of the load. In this strategy, we only use the core PU and not the hyper-threads PU, so every process would not share his core usage (all the cores are being used). The third strategy (S3) consists also on splitting the workload between the two sockets, but considering the usage of the hyper-threads on each core—*i.e.*, half of the cores are being used over the two CPU. Figure 3.7 reports on the energy consumption of the three strategies when running the benchmark `CG` on the cluster `Dahu`. We can notice the big difference between these three execution modes that we obtained only by changing the PU pinning method (that we acknowledged with more than 100 additional runs over more than 30 machines and with the benchmarks `LU` and `EP`). For example, S2 is the least power consuming strategy. We argue that the reason is related to the isolation of every process on a single physical core, reducing the context switch operations. In the first and third strategy, 32 processes are being scheduled on 16 physical cores using the hyper-threads PU, which will introduce more context switching, and thus more energy consumption.

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

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

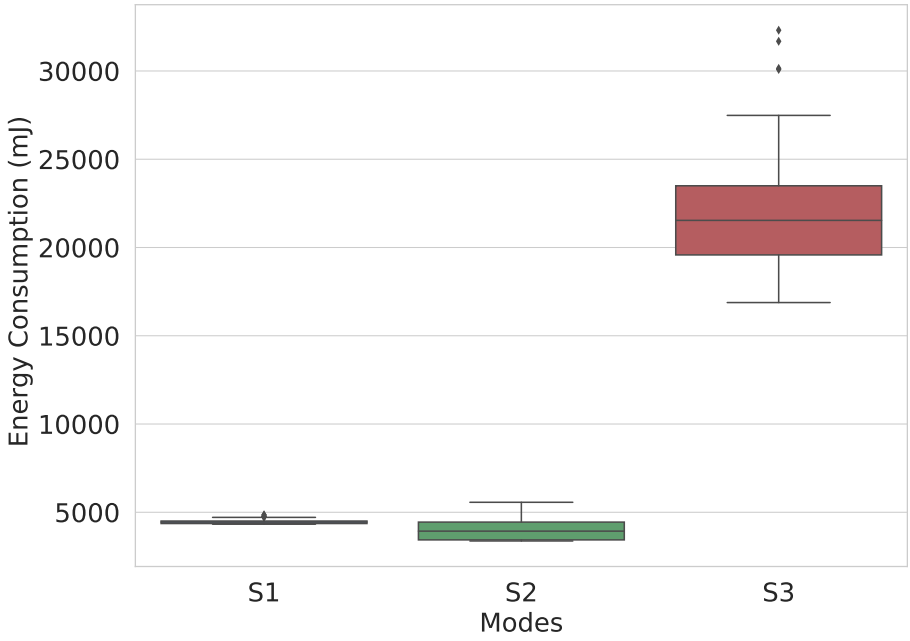


Figure 3.7: Energy variation considering the three cores pinning strategies at 50 % workload

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.

Our hypothesis regarding the worst results that we observed when using the third strategy is the recurrent context switching, added to the OS scheduling that could reschedule processes from a socket to another, which invalids the cache usage as a process can not take profit of the socket local L3 cache when it moves from a CPU to another (cf. Figure 3.3).

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

Table 3.2 reports on additional aggregated results for the STD comparison on four other nodes of the cluster Dahu at 50 %, with the benchmarks LU, CG and EP. In fact, the CPU usage strategy S1 is by far the experimentation mode that gave the least variation. The STD is almost 5 times better than the strategy S2, but is up to 10 % more energy consuming ($Mean_{S1} = 4469mJ$, $Mean_{S2} = 4016mJ$). On the other hand, the strategy S3

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

is the worst, where the energy consumption can be up to 5 times higher than the strategy S2 ($Mean_{S2} = 4016mJ$, $Mean_{S3} = 21645mJ$) and the variation is much worst (30 times compared to the first strategy). These results allow us to have a better understanding of the different processes-to-PU pinning strategies, where isolating the workload on a single CPU is the best strategy. Using the hyper-threads PU on multiple sockets seems to be a bad recommendation, while keeping the hyper-threading enabled on the machine is not problematic, as long as the processes are correctly pinned on the PU. Our experiments show that running one hyper-thread per core is not always the best to do, at the opposite of the claims of [**?**].

Processes Threshold To answer the second question regarding the evolution of the energy variation at different levels of CPU usage, we varied the used PU's count to track the EV evolution. Figure 3.8 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 clearly observe, the variation is increasing along with the number of processes. When running only 2 processes, turning off the C-states reduces the STD up to 6 times, but consumes 20 % more energy ($Mean_{C-states-on} = 10,334mJ$, $Mean_{C-states-off} = 12,594mJ$). This variation is 4 times lower when running 4 processes and almost equal to the C-states on mode when running 8 processes. In fact, running more processes implies to use more CPU cores, which reduces the idle cores count, so the cores will more likely stay at a higher consumption state even if the C-states mechanism is on.

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

We note that disabling the C-states is not recommended in production environments, as it introduces extra energy consumption for low workloads (around 50 % in our case for a single process job). However, our goal is not to optimize the energy consumption, but to

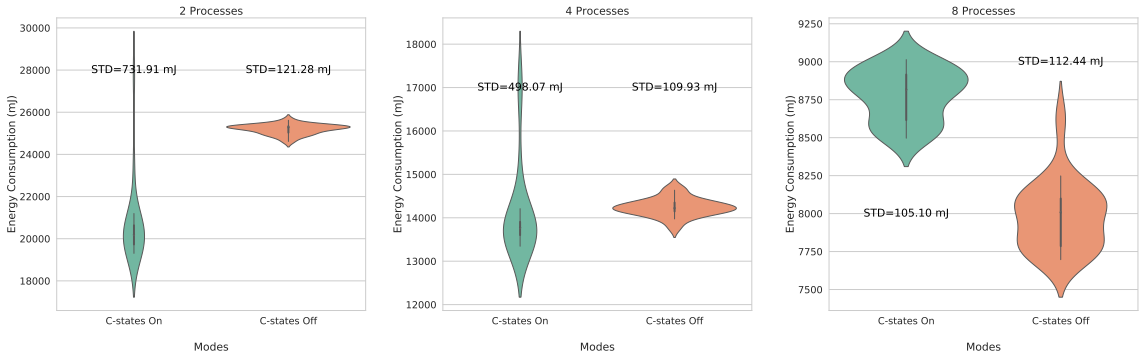


Figure 3.8: C-states effect on the energy variation, regarding the application processes count

minimize the energy variation. Thus, disabling the C-states is very important to stabilize the measurements in some cases when the variation matters the most. Comparing the energy consumptions of two algorithms or two versions of a software systems is an example of use case benefiting from this recommendation.

Turbo Boost The Turbo Boost—also known as *Dynamic Overclocking*—is a feature that has been incorporated in Intel CPU since the Sandy Bridge micro-architecture, and is now widely available on all of the Core i5, Core i7, Core i9 and Xeon series. It automatically raises some of the CPU cores operating frequency for short periods of time, and thus boost performances under specific constraints. When demanding tasks are running, the operating system decides on using the highest performance state of the processor.

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

The experiment we conducted showed that disabling the Turbo Boost does not exhibit any considerable positive or negative effect on the energy variation. Table 3.3 compares the STD when enabling/disabling the Turbo Boost, where the columns are a combination of workload and benchmark. In fact, we only got some minor measurements differences when switching on and off the Turbo Boost, and where in favor or against the usage of the Turbo Boost while repeating tests, considering multiple nodes and benchmarks. This behavior is mainly

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

related to the *thermal design power* (TDP), especially at high workloads executions. When a CPU is used at its maximum capacity, the cores would be heating up very fast and would hit the maximum TDP limit. In this case, the Turbo Boost cannot offer more power to the CPU because of the CPU thermal restrictions. At lower workloads, the tests we conducted proved that the Turbo Boost is not one of the main reasons of the energy variation. In fact, the variation difference is barely noticeable when disabling the Turbo Boost, which cannot be considered as a result regarding the OS activity and the measurement error margin. We cannot affirm that the Turbo Boost does not have an impact on all the CPU, as we only tested on two recent Xeon CPU (clusters Chetemi and Dahu). We confirmed our experiments on these machines 100 times at 5 %, 25 %, 50 % and 100 % workloads.

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

RQ 3: Operating System

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

The first way to do is to evaluate the OS idle activity consumption, and to compare it to a low workload running job. Therefore, we ran 100 iterations of a single process benchmark EP, LU and CG on multiple nodes from the cluster Dahu, and compared the energy behavior of the node with its idle state on the same duration. The aggregated results, illustrated in Figure 3.9, 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

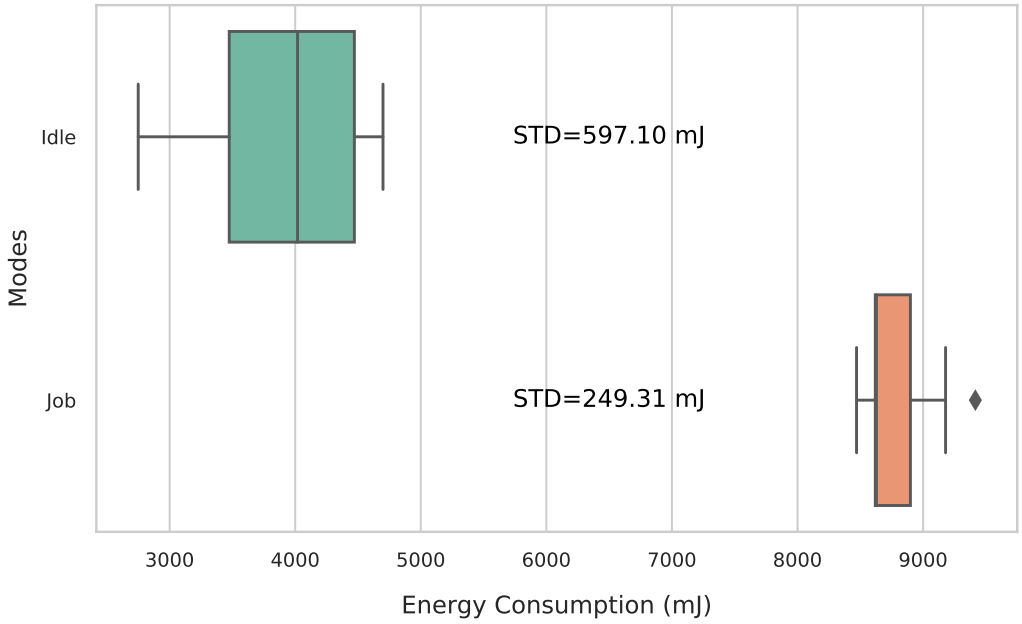


Figure 3.9: OS consumption between idle and when running a single process job

way more important than when a test was running, even if it is a single process test on a 32-cores node. This result shows that OS idle consumption varies widely, due to the lack of activity and the different CPU frequencies states, but it does not mean that this variation is the main responsible for the overall energy variation. The OS behaves differently when a job is running, even if the amount of available cores is more than enough for the OS to keep his idle behavior when running a single process.

Inspecting the OS idle energy variation is not sufficient to relate the energy variation to the active job. In fact, the OS can behave differently regarding the resource usage when running a task. To evaluate the OS and the job energy consumption separately, we used the PowerAPI toolkit. This fine-grained power meter allows the distribution of the RAPL global energy across all the Cgroups of the OS using a power model. Thus, it is possible to isolate the job energy consumption instead of the global energy consumption delivered by RAPL. To do so, we ran tests with a single process workload on the cluster Dahu, and used the PowerAPI toolkit to measure the energy consumption. Then, we compared the job energy consumption to the global RAPL data. We calculated the Pearson correlation [2] of the energy consumption and variation between global RAPL and PowerAPI, as illustrated in Figure 3.10. The job energy consumption and variation are strongly correlated with the global energy consumption and variation with the coefficients 93.6 % and 85.3 %, respectively. However, this does not completely exclude the OS activity, especially if the jobs have tight

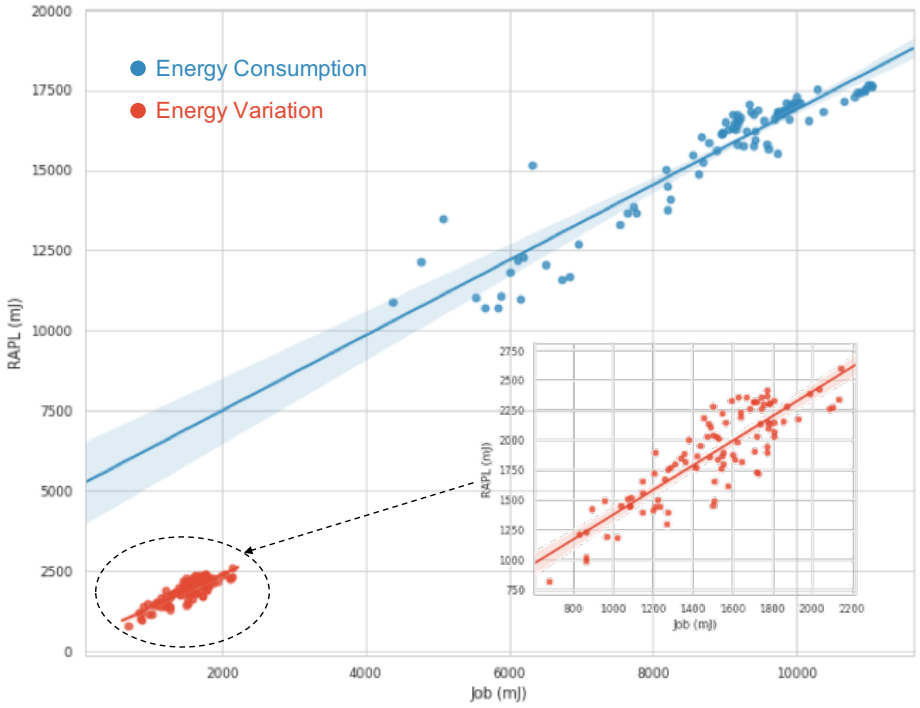


Figure 3.10: The correlation between the RAPL and the job consumption and variation

interaction with the OS through the signals and system calls. This brings a new question on whether applying extra-tuning on a minimal OS would reduce the variation? As well as what is the effect of the Meltdown security patch—that is known to be causing some performance degradation [**? ?**]—on the energy variation?

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

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 %

difference, due to many factors that can affect it as the CPU temperature or the measurement errors.

Speculative Executions Meltdown and Spectre are two of the most famous hardware vulnerabilities discovered in 2018, and exploiting them allows a malicious process to access others processes data that is supposed to be private [**? ?**]. They both exploit the speculative execution technique where a process anticipates some upcoming tasks, which are not guaranteed to be executed, when extra resources are available, and revert those changes if not. Some OS-level patches had been applied to prevent/reduce the criticality of these vulnerabilities. On the Linux kernel, the patch has been automatically applied since the version 4.14.12. It mitigates the risk by isolating the kernel and the user space and preventing the mapping of most of the kernel memory in the user space. Nikolay *et al.* have studied in [74] the impact of patching the OS on the performance. The results showed that the overall performance decrease is around 2–3 % for most of the benchmarks and real-world applications, only some specific functions can meet a high performance decrease. In our study, we are interested in the applied patch's impact on the energy variation, as the performance decrease could mean an energy consumption increase. Thus, we ran the same benchmarks LU, CG ad EP on the cluster Dahu with different workloads, using the same OS, with and without the security patch. Table 3.5 reports on the STD values before disabling the security patch. A minus means that the energy varies less without the patch being applied, while a plus means that it varies more. These results help us to conclude that the security patch's effect on the energy variation is not substantial and can be absorbed through the error margin for the tested benchmarks. In fact, the best case to consider is the benchmark LU where the energy variation is less than 10 % when we disable the security patch, but this difference is still moderate. The little performance difference discussed in [**? ?**] may only be responsible of a small variation, which will be absorbed through the measurement tools and external noise error margin in most cases.

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

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 %

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

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

RQ 4: Processor Generation

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

Table 3.6 also shows the aggregated energy variation of the different generations of nodes for the benchmarks LU, CG and EP. The results attest that the latest versions of CPU do not necessarily cause more variation. In the experiments we ran, the nodes from the cluster Paranoia tend to cause more variation at high workloads, even if they are from the latest generation. While the Skylake CPU of the cluster Dahu cause often more energy variation than Chetemi and the Ecotype Broadwell CPU. We argue that the hypothesis "*the energy consumption on newer CPU varies more*" could be true or not depending on the compared generations, but most importantly, the chips energy behaviors. On the other hand, our experiments showed the lowest energy variation when using the Ecotype CPU, these

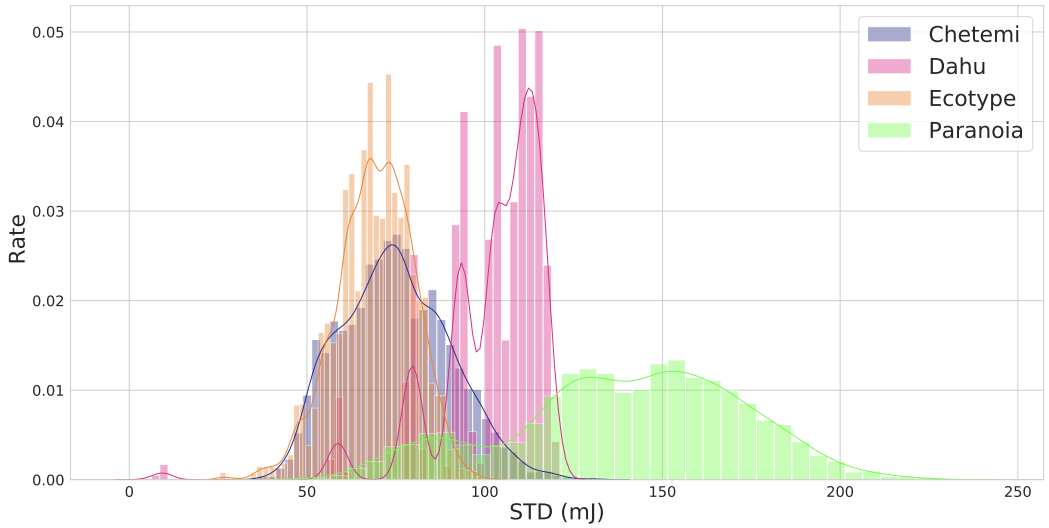


Figure 3.11: Energy consumption STD density of the 4 clusters

CPU are not the oldest nor the latest, but are tagged with "L" for their low power/TDP. This result rises another hypothesis when considering CPU choice, which implies selecting the CPU with a low TDP. This hypothesis has been confirmed on all the Ecotype cluster nodes, especially at low and medium workloads.

Figure 3.11 is an illustration of the aggregated STD density of more than 5,000-random values sets taken from all the conducted experiments. This shows that the cluster Paranoia reports the worst variation in most cases, and that Ecotype is the best cluster to consider to get the least variations, as it has a higher density for small variation values.

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

3.3.4 Experimental Guidelines

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

Table 3.7 gives a proper understanding of known factors, like the C-states and its variation reduction at low workloads. However, it also lists some new factors that we identified along

Table 3.7: Experimental Guidelines for Energy Variations

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

the analysis we conducted in Section, such as the results related to the OS or the reboot mode. Some of the guidelines are more useful/efficient for specific workloads, as showed in our experiments. Thus, qualifying the workload before conducting the experiments can help in choosing the proper guidelines to apply. Other studied factors are not been mentioned in the guidelines, like the Turbo Boost or the Speculative execution, due to the small effect that has been observed in our study.

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

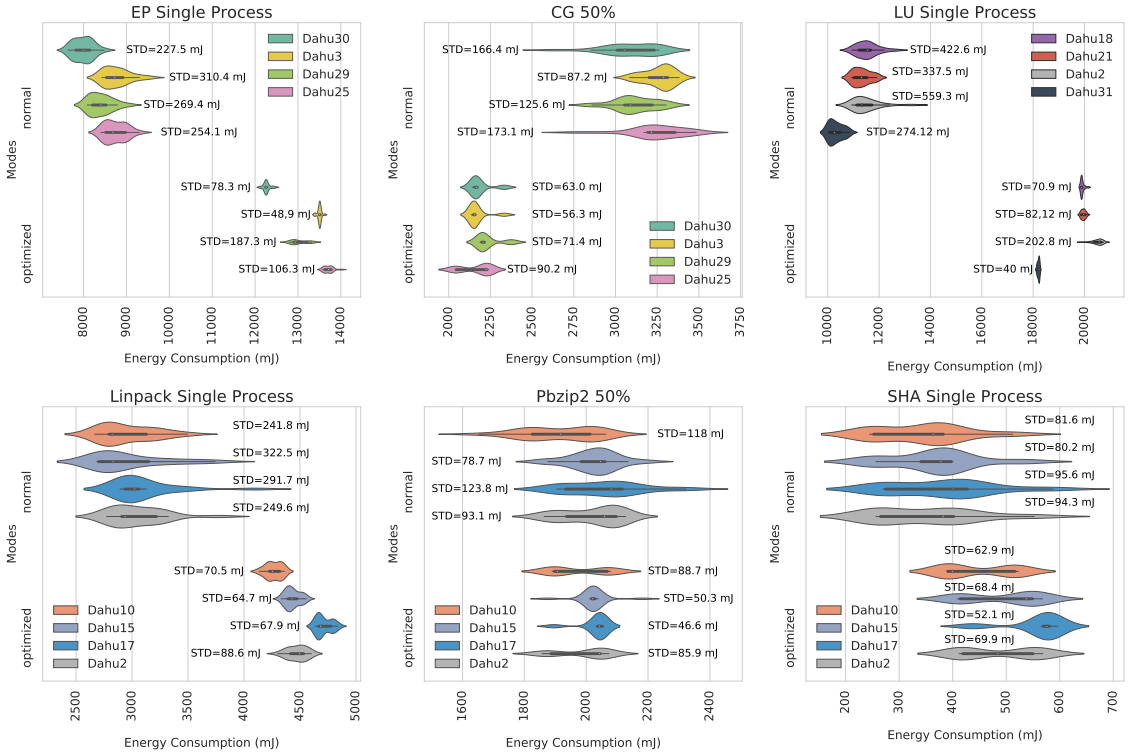


Figure 3.12: Energy variation comparison with/without applying our guidelines

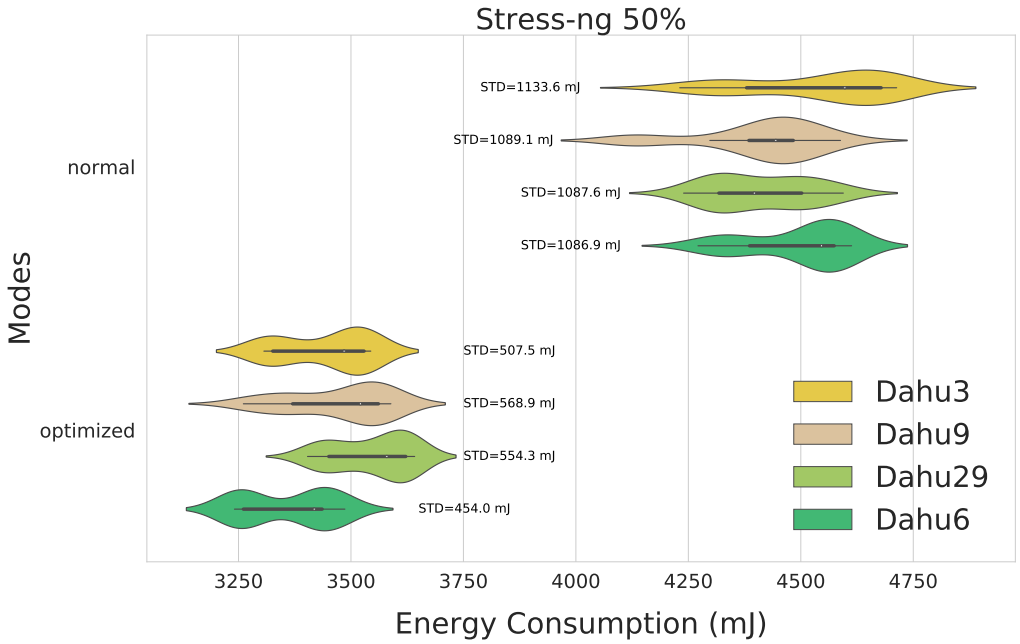


Figure 3.13: Energy variation comparison with/without applying our guidelines for STRESS-NG

Figure 3.12 and 3.13 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 ($Mean_{LU-normal-Dahu2} = 11,500mJ$, $Mean_{LU-optimized-Dahu2} = 20,508mJ$). For medium workloads, the STD, and thus variation, is up to 100 % better for the benchmark CG, 20–150 % better for the `pbzip2` application and up to 100% for STRESS-NG. We note that the optimized version consumes less energy thanks to an appropriate core pinning method.

Figures 3.12 and 3.13 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 to identify more relevant mediums and medians, and then perform accurate comparisons between the nodes variation. Even though, using the same node is always better, to avoid the extra inter-nodes variation and thus improve the stability of measurements.

3.3.5 Threats to Validity

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

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

3.3.6 Conclusion

In this part, we conducted an empirical study of controllable factors that can increase the energy variations on platforms with some of the latest CPU, and for several workloads. We provide a set of guidelines that can be implemented and tuned (through the OS GRUB for example), especially with the new data centers isolation trend and the cloud usage, even for scientific and R&D purposes. Our guidelines aim at helping the user in reducing the CPU energy variation during systems benchmarking, and conduct more stable experiments when the variation is critical. For example, when comparing the energy consumption of two versions of an algorithm or a software system, where the difference can be tight and need to be measured accurately.

Overall, our results are 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. The aim of our work is to be able to tame and mitigate this variability along controlled experiments. We studied some previously discussed aspects on some recent CPU, considered new factors that have not been deeply analyzed to the best of our knowledge, and constituted a set of guidelines to achieve the variability mitigating purpose. Some of these factors, like the

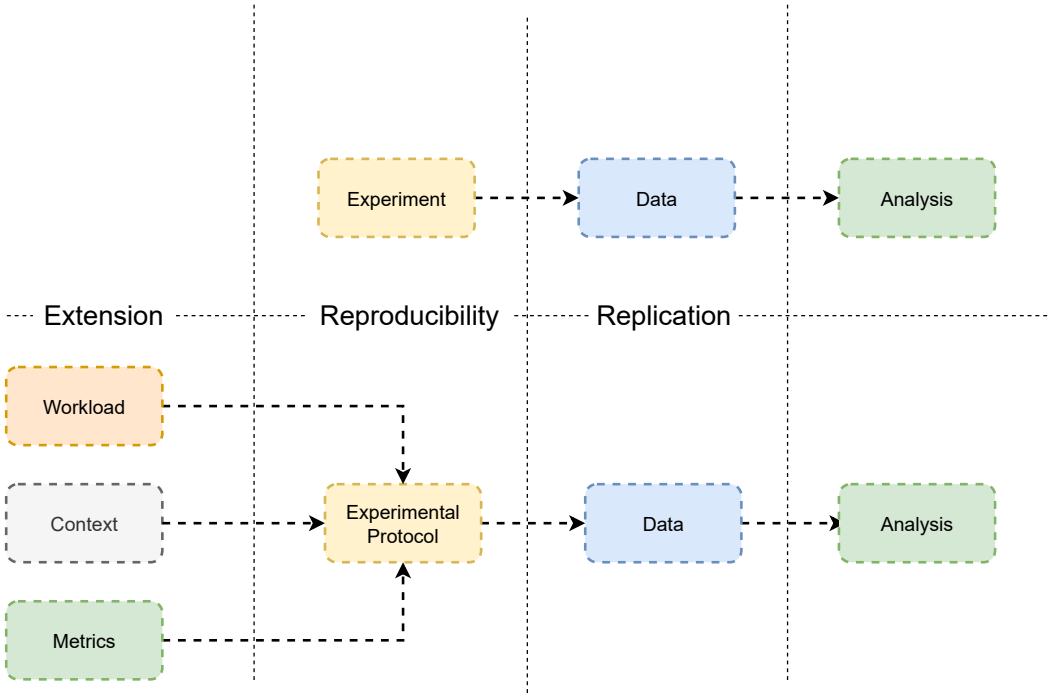


Figure 3.14: Benchmarking protocol

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.

We believe that our approach can also be used to study/discover other potential variability factors, and extend our results to alternative CPU generations/brands. Most importantly, this should motivate future works on creating a better knowledge on the variability due to CPU manufacturing process and other factors.

3.4 Extension

In computer science, things are changing fast, which leads to a raise of obsolete results. Specially when it comes to Comparison studies,One can't cover all the candidates. Moreover, between the initial experiment and the published results there might be new candidates and an evolution of others. Therefore we Propose new creterioin for a successful experiment Extension . In our content Extension means the ability to provide the necessary tools to not only Repeat the experiment but to be able to add extra candidates, Workloads or metrics. For most of our experiments we use docker images for each candidate. This allow us to change the context of the experiment without impacting the protocol.

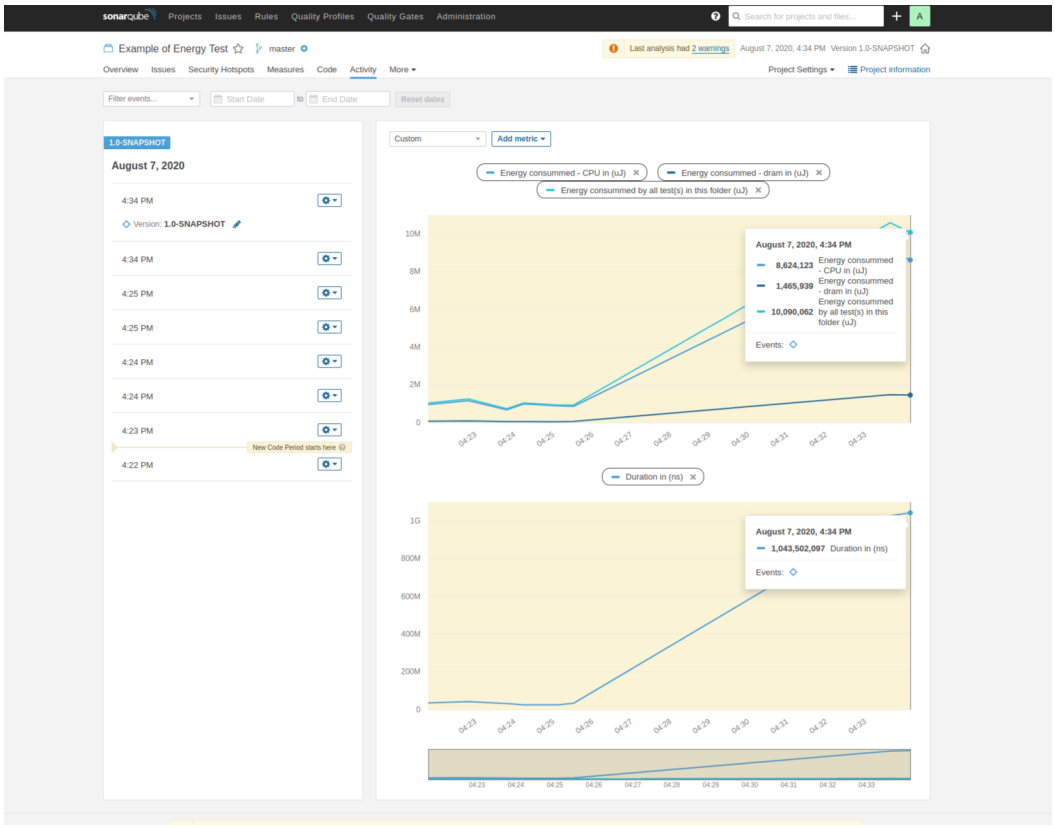


Figure 3.15: Example of the Junit Sonar Plugin

3.5 Perspectives

By the end of this study we have gathered enought guidelines to make the tests more reproducible, accurate. We created a set of new tests named **energy tests** which are more similar to performance tests. Thanks to the work of two interns [mamadou and adrien] we created a CI/CD plateform to measure the energy consumption of Java projects and we could track the evolution of the this energy accross different stages of the project. In the figure below we see an example of this plugin. For more details please visit the gitlab repository ... add link.

Chapter 4

Energy footprint of programming languages

In this chapter, we discuss the impact of the choice of the programming language on the energy consumption of the resulting software. To do so, we suggest to start with the general micro benchmarking and see how each programming languages interact with the CPU/Memory.

The ultimate goal of this chapter is to provide a guideline on which programming language the developers should chose based on the charecteristics of the project in order to minimize the energy foot print of their product. No answer is evidedent for a such question. However, we can extract some feartures of each programming langues such as

- performance
- community support
- scalability
- energy consumptiom
- memory usage
- etc

first we will start but analysing the behaviour of the general purpose programming langauges with some micro benchkaks, principally for the CLBG game and others from rosetta code base

As we have seen in the previous chapter, one of the most important feature of a test is to be **representative**. Therefore, we extend this study to some real-life use case, and the following sections below provide two case studies.

4.1 Remote Procedure Call Frameworks

introduction

With the emergence of cloud technologies, many protocols wanted to take the lead. Nowdays, most of the architectures are based on multi-services and microservices. And, to have higher versatility of developers multiple companies choose to be open to different programming languages. Basically, it would be more efficient if we take advantage of each programming language to satisfy a specific need. However, the challenge nowadays is to make the bridge between those platforms. We have many initiatives, such as OpenAPI, that try to create a taxonomy for RESTful APIs. Other approaches implement all the different interfaces of the protocol by themselves, such as RPC.

4.1.1 Research Questions

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

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

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

4.1.2 Experimental protocol

measurement context

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

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

metrics

Energy measurements To report on the energy consumption, we used HwPC sensor [TOCITE], which is based on Intel RAPL technology, one of the most accurate tools to measure the energy consumption of the CPU and DRAM [TOCITE]. For better accuracy, we ran the HwPC sensor with a frequency of 10 Hz, and we used the same machine for all the experiments in order to reduce the variability [TOCITE].

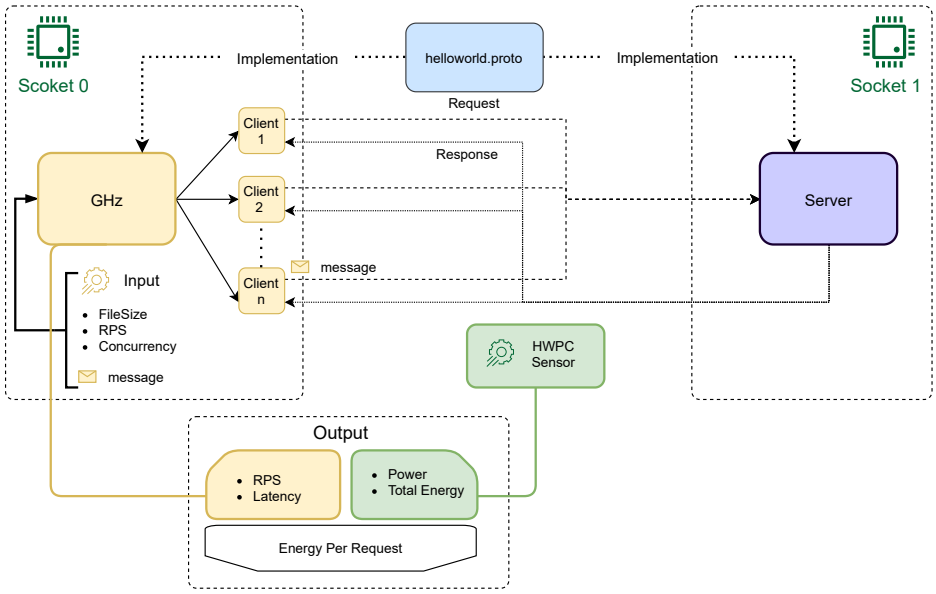


Figure 4.1: Experimental software architecture

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

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

workload

The purpose of the experiment is to analyse the behaviour of different GRPC implementations. therefore we have two kind of worlkoads

number of clients : this is the number of concurrent clients that we want to test. It is automatically handled with the ghz client

payload : the size of each request . this varries from 50 Bytes up to 10 MB.

The client consumes the protocol description which can be found if 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 created through a docker image, which will allow us to add new implementations easily.

Extension

: for the sake of extedind the experement we provide a github repository [2] that containes the implementation of the experement. Adding **candidates** 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 easly by adding new files in the folder **payload**.

4.1.3 Results and findings

[RQ 1:] How do RPC implementations react to the size of the data ? The purpose of this question is to study the behaviour of the 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, which results 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 limit for each request. Therefore, our boundary condition is only the number of requests received by the server. For this experiment, we investigate 4 observable variables :

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

1. the average power consumption during the the process: this will indicate the overall behaviour of the server in working mode for long durations,
2. the tail latency for the 99th percentile, which indicates how performant is the server,
3. the average number of requests per second, which indicates the average number of clients that the server can handle,
4. the average energy cost of a single request: unlike the first indicator, this one shows how green is the implementation taking performance into consideration.

The above figure depicts the overall behaviour of each framework based on the size of request (payload). For each framework, we can distinguish three modes, and they all depend on the payload size:

1. Stress free mode when the server has enough resources to satisfy the requests because 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, however the server can still manage to handle them, and here where we can see a change in the energetic and performance behaviour,
3. Broken state mode when the requests are much heavier and the server break—like 10 MB.

Stress free mode

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

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

Escalation mode

In this mode, the behaviour of the server depends on the payload. We observe three behaviours:

1. Drop in performances without an increased power, such as .Net core, Java micronaut, Crystal, and Dart. In this case, the server keeps using the same resources, and sometimes less, because it takes more time to handle the less requests. This class of languages tend to be the most energy consuming when it comes to the cost per request,
2. Increase in power without affecting the performances: such as Go, .Net. The energy consumption of a single request, is affected slightly but still increase,
3. Increase in power and drop in performances: Despite the increase of the power consumption, the server becomes slightly slower, which increases the cost of 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 kept scaling despite the lack of performances compared to other compiled languages (Go, CPP).

Broken state mode

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

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

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

Power behaviour

Based on the heatmap, we can distinguish two modes:

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

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

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

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

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

Stressed mode Although the same classes remained the same, not all the languages had the same evolution and here we can clearly observe a correlation with the category of the programming language rather than the implementation itself. We can clearly highlight that VM-based languages have a significant increase (double) in the average power consumption after they receive more than 100 concurrent clients. Except PHP, all the interpreted and compiled languages preserved their energetic behaviour. Our hypothesis points to the JIT, since it compiles the code and makes it run faster, hence stressing the CPU. A interesting behaviour has been noticed for the GraalVM: the decrease of energy consumption when increasing the number of the clients. This is related to the drop of the performances, which was probably due to the bottleneck situation where the GraalVM could not handle more than 100 clients simultaneously.

Performance Behaviour

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

- Satisfaction ratio: how many requests have been satisfied among the total requests,

- Request Per Seconds: The number of the requests that have been answered from the server,

- Tail Latency at 99%: one of the best metrics to evaluate the performances of a server.

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

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

Tail Latency The increase in the number of requests per second, does not necessarily mean a lower latency. As one can notice in Table **??**, until the 1000 clients, Go provides the least latency beside .Net. GraalVM provides the highest latency, on average. However, Dart tends to become slower when we increase the number of clients, until we pass the 600 simultaneous clients, and there it changes its behaviour, instead of satisfying most the requests it notify the clients directly that the server is saturated, hence a drop in satisfaction ratio, and an amelioration for the average latency.

Energy Per Request

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

4.1.4 Threads to validity

4.1.5 Conclusion

4.2 web Frameworks

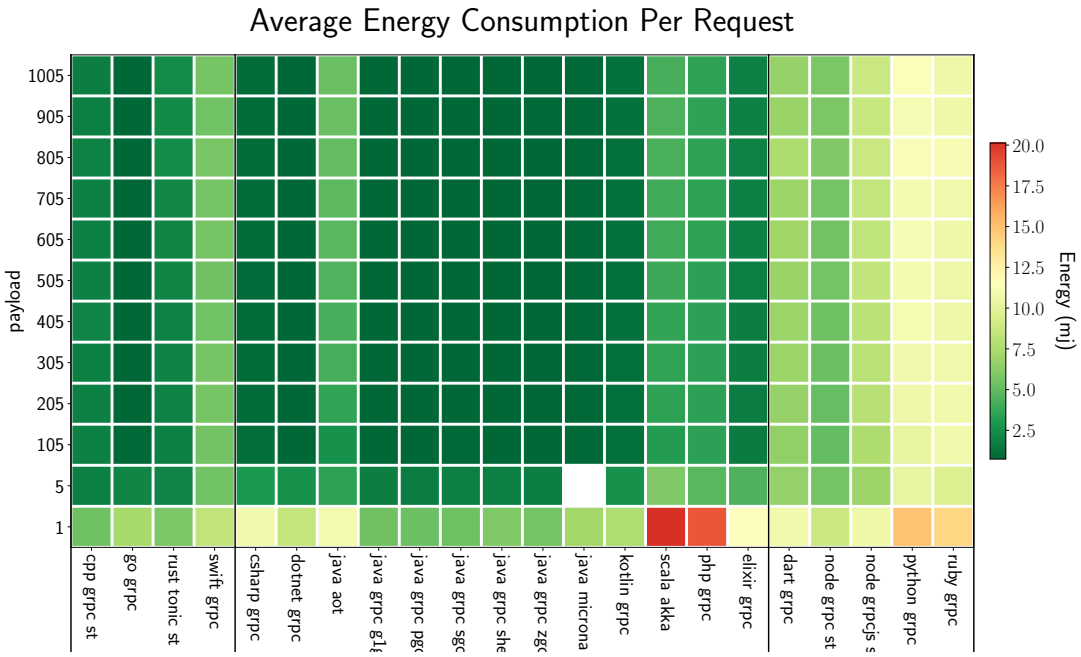


Figure 4.2: Experimental software architecture

4.2.1 Introduction

Nowadays, web applications are dominating online systems. From Google, to Facebook and others, web applications are widely deployed across organizations and continuously accessed by end-users, both for their personal and professional daily tasks. In practice, the development of these web applications heavily rely on a wide ecosystem of *web frameworks*, which are intended to ease and foster the development process. However, once deployed, the applications developed with such web frameworks do not exhibit the same performances, as reported by the *Web Framework Benchmarks* periodically published by the TECHEMPOWER company.[3] Thanks to such benchmarks, developers can take informed decisions on the most performing technology to adopt to implement their web applications. Unfortunately, one can regret that developers and benchmark providers focus mostly on popularity and performance criteria when picking a web framework, with less considerations for the resource consumption implications of their choice. This is all the more regrettable that cloud providers are more and more adopted by developers to host these web applications. While cloud providers offer a convenient elastic provision of resources to scale according to application requirements, this convenience may induce critical cost for their business.

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

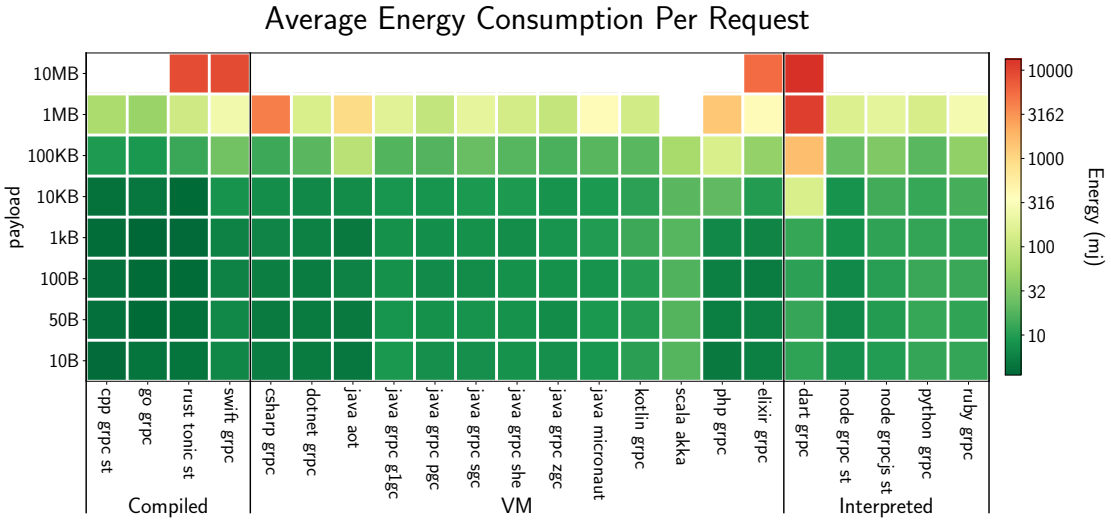


Figure 4.3: Experimental software architecture

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

Given this context, this paper intends to contribute to 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 provide the best performances, 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 reduce worldwide carbon emissions of ICT.

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

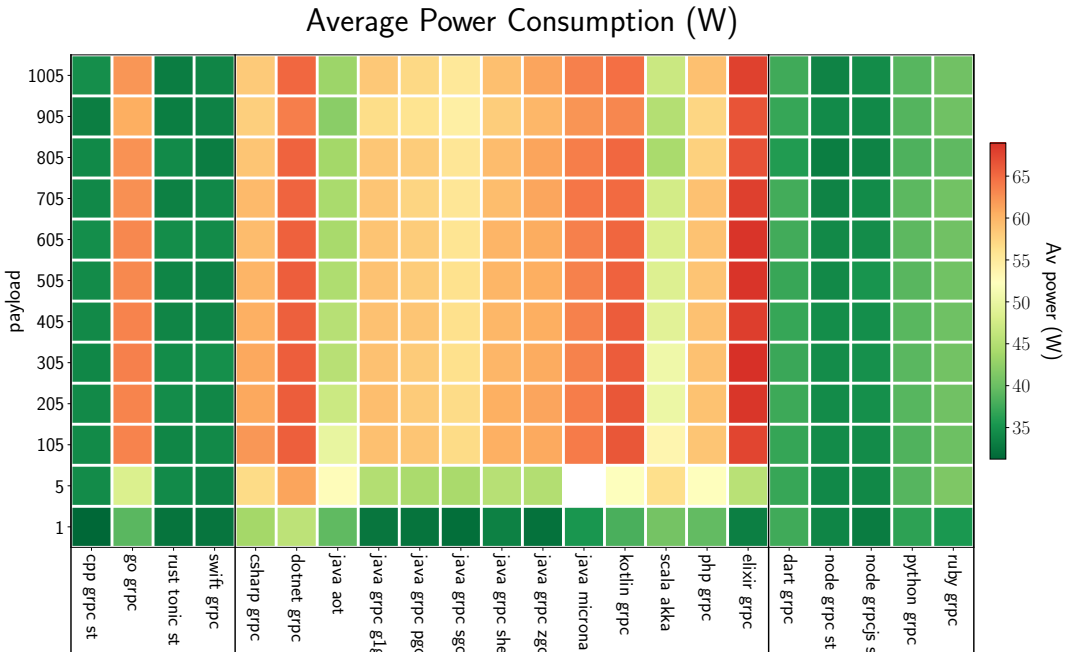


Figure 4.4: Experimental software architecture

guidelines to can help developers to pick the most energy efficient web frameworks according to their requirements.

The remainder of this paper is organized as follows.

4.2.2 Comparaison between web frameworks

Many studies have been conducted to compare the performance of web frameworks.

We can cite [**?**] where they made a comparison between two of the most famous java frameworks, Play and Spring, or the work of [**?**] when they compare different PHP frameworks using 6 creterions intrinsic durability, industrialized solution, technical adaptability, strategy, technical architecture, and Speed, In the previous paper we find that all those 6 creterions have their own wait when it comes to choose which framework to take for a project. In our study we want to push a 7th criterion that impacts the economic outcome of the project.

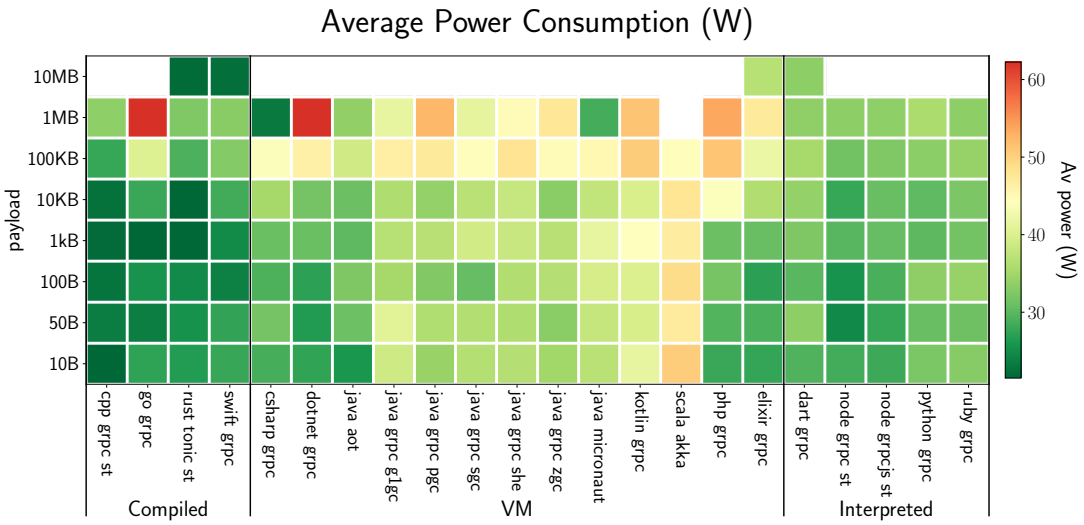


Figure 4.5: Experimental software architecture

4.2.3 energy efficiency in software engineering

In their paper [**?**] the authors studied the impact of programming languages on energy, time, and Memory by using the CLBG benchmark, where they executed 10 different benchmarks [add citation] accross 27 wel-know programming languages [add reference] the work of the authors was an extension of the a research marled by [add sitation[6 in the authers paper]]. we continued their work with measuring the impact of programming languages choice in real life application instead of microbenchmark

Irene et al.[55] investigated the impact of web servers on energy when handleling web applications . they analysed 7 applications executed within 4 serveces with 38 different scenarios . the authors showed that the energy greatly depends on the web server , howerver the impact of the application may influence this energitical behaviour of the server.

In their approach they used measured the energy consumption during the integration tests and for us we measure with a simulation of a website when there is a client in another machine that requests the website, therefore we isolate the energy consumption of the server from the the client's one.

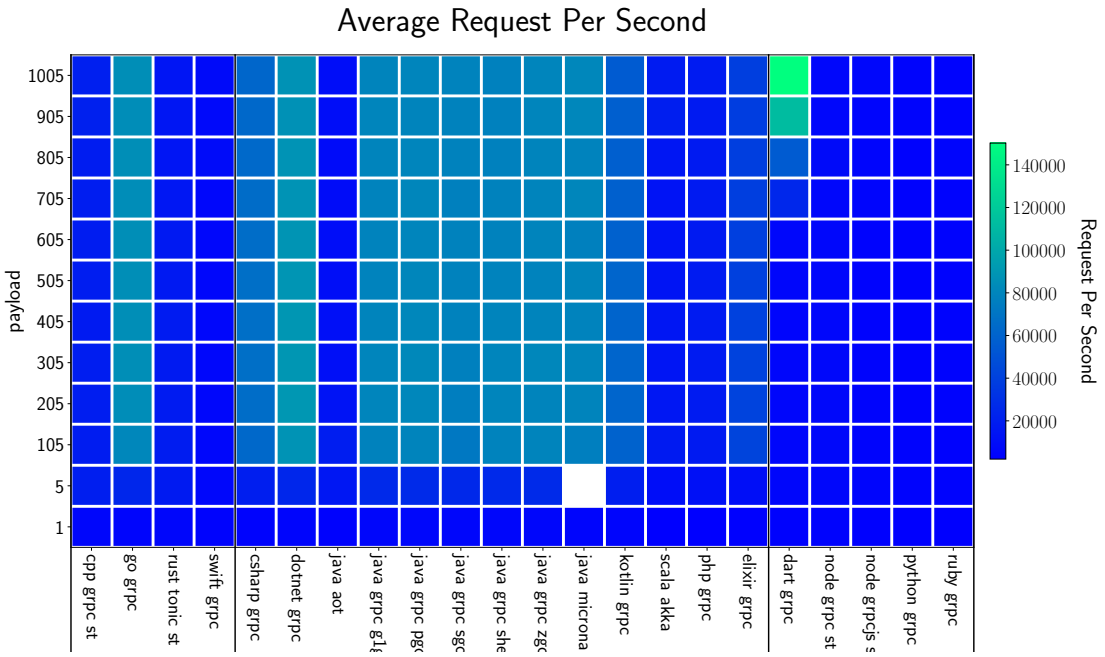


Figure 4.6: Experimental software architecture

Other works have been done on the client side, as an example [**?**] as they concluded that theres is a variation among the different websites and the impact of the browser on this energy.

4.3 Experimental Protocol

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

4.3.1 Measurement context

The purpose of this exmeperiment is to highlight the impact of the technology stack of developping a website on the energy consumption whule its execution.

Table **??** highlights the number of frameworks used in the experiment and they passed each category of tests. As we see in the table below, some of the frameworks worked on certain conditons while they failed on other tests such as nickel from rust, while nickel might be one of the greenest rust framework, it doens't work with databases. Therefore, it can be

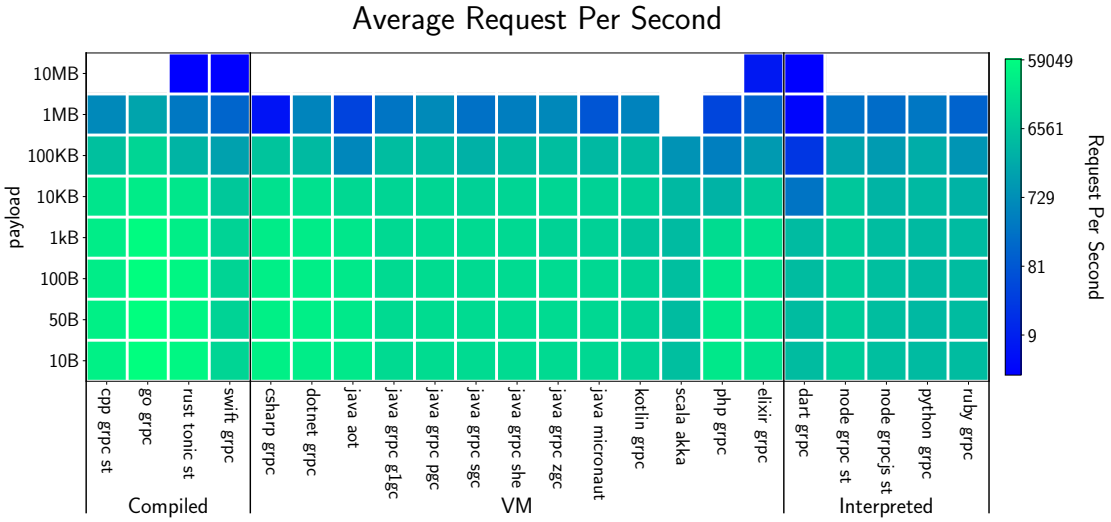


Figure 4.7: Experimental software architecture

used within all the situations, but if the website doesnt interact with a database then it might be the best choice.

many reasons are behind the failure, either there were no implementation or just there were some errors while handling the request.

TODO : we decided to skip the idle part in the validation test since it is not relevant

- orchestrator : the part that is responisble for creating docker images, selecting the benchmarks and launching the tests;
- web server : or the system under test (SUT) is the machine responsible for launching the framework by means of the pre-installed powermeter;
- database server : a offers the same data base that will be used by all the frameworks during the tests
- client machines : To avoid the bottleneck on the client's side, client requests are sent from another machine (one or many) that simulates hundreds of concurrent connections to the framework.

Table 4.1: the number of framework passed per test

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	1203

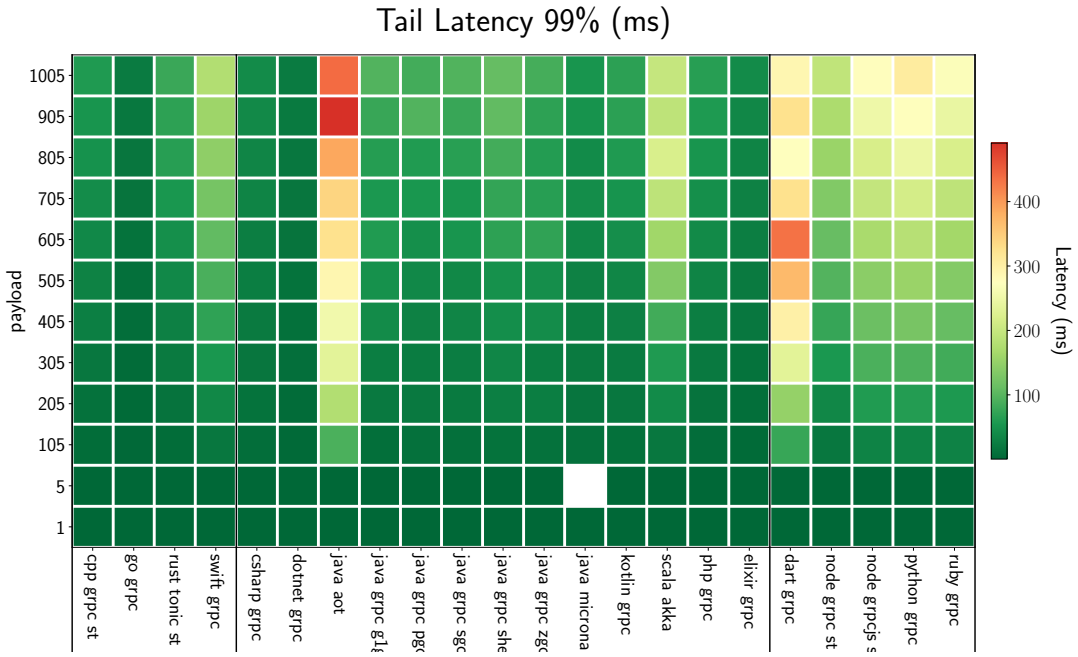


Figure 4.8: Experimental software architecture

- recorder : the party responsible for collecting the power data from the SUT and the performance metrics collected by the clients ***TODO: add link to the measurement process***

The tests have been executed in machines from the cluster chetemi of the ***ADD Reference: Grid500*** plateform . ***TODO: add hardware description***

Note It has been proven in cite that docker does not influence on the energy cosumption. Thus, using its containers and isolation allow to avoid any alteration of the operation system after testing one benchmark and to take advantage of its reproducibility.

4.3.2 workload

To mesure compare the energy consumption and performance effeciency between mutiple frameworks, each framework used to implement the samewebsite answering the same urls and requesting the same database. Then we run the same algorthim for all impelentations.

1. lunch the framework
2. wait for 20s for the warmup

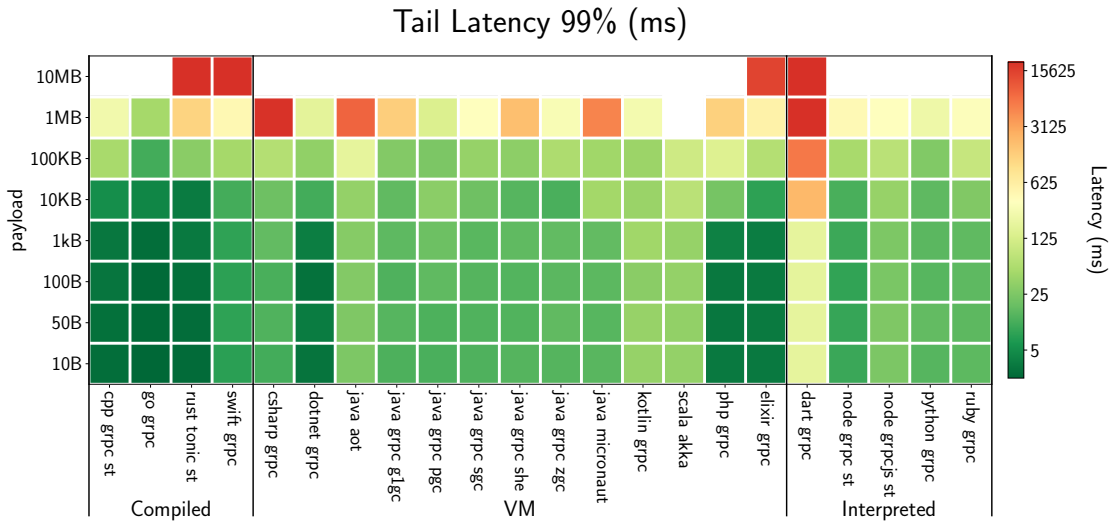


Figure 4.9: Experimental software architecture

3. measure the average power when the framework is in idle state
4. using multiple clients. we send the same request in simultaneously during 20s
5. increase the number of parallel request
6. measure the enrgy during this execution
7. change the request type
8. repeat from 3 rd step

the section will detail each type of experiments and the purpose behind it giving some examples of the expecteed response

Test categories

We have 7 categories of tests :

Idle In this test we measure the idle energy consumption of the framework: this reflects the average energy consumption of an application during the periods when we have no clients. For example, a company website beyond working hours or a small website at night.

Single query During this test, each request is processed by fetching a single row from a simple database table. That row is then serialized as a JSON response.

Multiple queries this test aims to test the behaviour of a framework when it process multiple entries from the database. Therefore each request is processed by fetching multiple rows from a simple database table and serializing these rows as a JSON response. In this case we use the maximum number of clients which is 512512

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

Update queries This test exercises database writes. Each request is processed by fetching multiple rows from a simple database table, converting the rows to in-memory objects, modifying one attribute of each object in memory, updating each associated row in the database individually, and then serializing the list of objects as a JSON response. same as the multiple queries we use here the maximum number of clients which is 512512

The response is analogous to the multiple-query test.

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

JSON Serialization In this test,each response is a JSON serialization of a freshly-instantiated object that maps the key message to the value "Hello, World!"

For each one of the above senarios we consider different level of stress table shows the different levels for each scenario.

To add extra scenarios one might implement a python class that handles the metadata of the workload, such as the query route, the query parameters and the excpected results.

Table 4.2: Stress levels for each scenario

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

4.3.3 Metrics

We focus on comparing the energitical behaviour of different frameworks in multiple scenarios

To measure the energy consumption of those frameworks, we launch each one for a fixed duration, then all the clients send multiple requests simultaneously. We calculate 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.

Runtime Measurements

- energy measurement : we use ***ADD Reference: powerapi***, a software powermeter to gather the runninng power 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 apporoach to estimate this energy. ***TODO: add formula*** after this we devide the calculated energy per number of responses during this phase we have 4 mertrics :
- Total cost of the energy during each period
- Total number of requests
- Average latency
- Average energy cost per request

Architecture

The aim is to compare the energy consumption of different frameworks. For this we consider the framework as a blackbox and we take into account the response to the 6 previous requests using the same database. In order to isolate the energy consumption of the framework the

test is run in a seperate machine with the minimum services and the powermeter. Figure 4.10 illustrates the architecture of our system

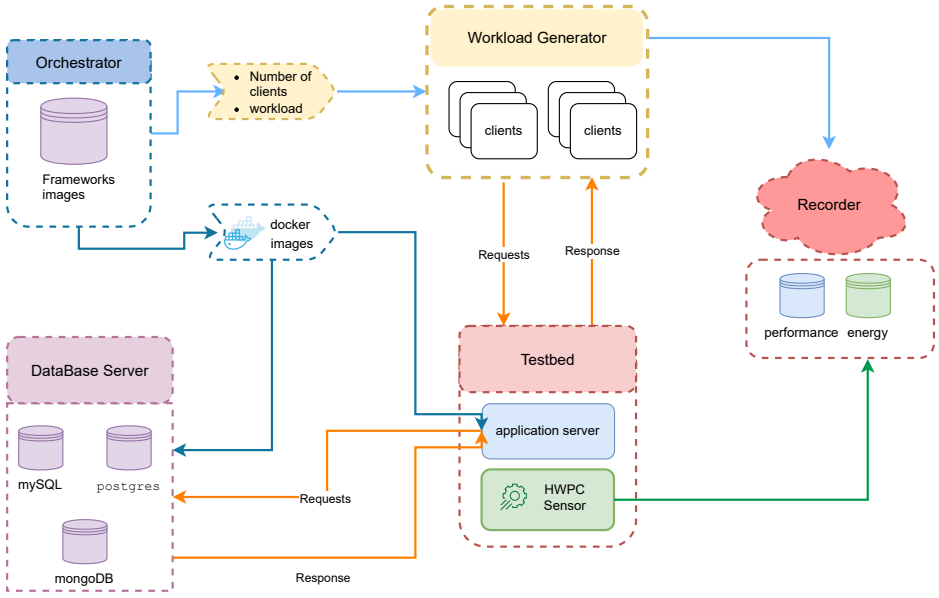


Figure 4.10: Architecture of the experiments

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

4.3.4 extesion

To follow the guidelines that we presented in chapter 2.6 we provide a github repository [4] where one can add extra **candidates** buy creating a new project using the option –new. then the they have just to fill the template and provide the docker image file.

to configure the **workload** we provide the option –concurrency-levels and –duration. As for the choice of the database it is added in the docker image.

4.4 finding and results

Overall we had 8750 test. and all can be found in the repository [5]. In this section we will use the previous results to answer the following questions

[4]https://github.com/chakib-belgaid/FrameworkBenchmarks
[5]https://github.com/chakib-belgaid/frameworks-benchmarks-results

- which programming language is the most efficient energy wise ?
- given a certain constraintes such as performance and number of clients, which stack perfromes the best ?

since most of the companies use the same stack, we wont be aggregating the results since it may lead to confusion. the best example, if we compare the average energy consumption for each programming langauge, this wont reflect in reality specially when we have two frameworks on the opposit sides of the spectrium.

4.5 tools and contribution

Chapter 5

The energy behavior of Python

5.1 Introduction

When it comes to developing software systems, one can notice that—except Perl—dynamic programming languages have taken over compiled languages along the last decade in terms of developers popularity (cf. Figure 5.1). However, it remains unclear if this class of dynamic programming languages can reasonably compete with compiled ones when it comes to power efficiency.

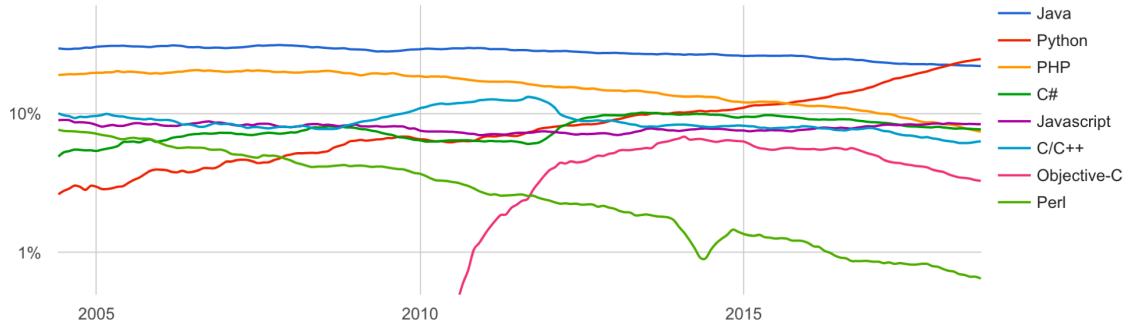


Figure 5.1: PYPL Popularity of Programming Languages [noa]

In particular, Noureddine *et al.* [**?**] in 2012, and then Pereira *et al.* [**?**] in 2017, conducted empirical power measurements on this topic, and both concluded that compiled programming languages overcome dynamic ones when it comes to power consumption. According to their experiments, an interpreted programming language like Python can impose up to a $7,588\,\%$ energy overhead compared to C [**?**] (cf. Figure 5.2). In this chapter, we therefore explore the oblivious optimizations that can be applied to Python legacy applications in order to reduce their energy footprint. As Python is widely adopted by software services deployed

in public and private Cloud infrastructures, we believe that our contributions will benefit to wide diversity of legacy systems and not only favorably contribute to reduce the carbon emissions of ICT, but also reduce their cloud invoice for the resources consumed by these services. More specifically, this chapter focuses on runtime optimizations that can adopted by developers to leverage the power consumption of Python applications. We start by studuing the impact of some of programmers choice such as the type of data structures or loops, on the global energy consumption of the execution code. then we discuss some other factors such as level of concurrency etc. later we we will talk about another non intrusive ways to optimize the energy consumption of this code. one kind of those optimizations includes alternative interpertersand some libraries that are dedicated to optimize the code without changing its structures such as *ahead-of-time* (AOT) compilation and *just-in-time* (JIT) libraries that are maintained by the community.

5.2 Motivation

5.2.1 python popularity

Nowdays, Python seems to attract a large community of developers interested in data analysis, web development, system administration, machine learning—according to a survey conducted in 2018 by JetBrains—[1]one can fear that the wide adoption of dynamic programming languages, like Python, in production 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 mostly for energy considerations. Rather, we believe that a better option consists in leveraging the strength of this rich ecosystem to promote energy-efficient solutions in order to improve the power consumption of legacy software systems.

5.2.2 python gluttony

the simplicity, flexibility and the clarity of Python code led this programming language to a huge success. Therefore, a larger community and a wide range of support and libraries that allowed it to touch several fields like data analyses, machine learning, natural language processing, web development, astrophysics etc.

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

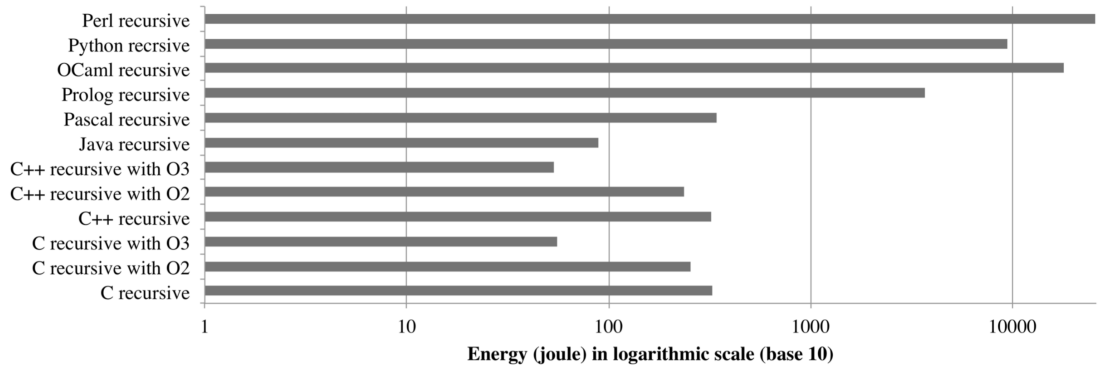


Figure 5.2: Energy consumption of a recursive implementation of Tower of Hanoi program in different languages [**?**]

However, and according to [**?**] and [**?**]. Python tends to be one more energy hungry . In figure 5.2, python consumes 30 times more than C or C++. The test was done on an implementation of hannoi test[2] 30 disks.

Python consumes a lot of energy mainly because it is slow in execution. Its Flexibility and simplicity caused it to drop off in performances, because Python gains its flexibility from being a dynamic language, therefore it needs an interpreter to execute its programs, which makes them much slower compared to the other ones that are written in compiled programming languages such as C and C++ or semi-compiled languages like Java.

As shown in Table 5.1, one can see that in the most of the test cases Python takes more time to execute -the only case that he wasn't the last one was in the benchmark regx-redux where he beaten Go-, and in some cases the gab was huge such as in the n-body test where python took around 100 times more than C++.

5.2.3 use cases

to reduce the energy consumption of python we started by targeting the main usage of this programming language, wich is revealed to be data sciences and web developpement. Figure 5.3 illustrate a study made by the jetbrain company on python developers [3]. In this survey, multiple multiple answers were accepted.

[2]https://en.wikipedia.org/wiki/Tower_of_Hanoi
[3]https://www.jetbrains.com/lp/python-developers-survey-2020[4]¸

	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

Table 5.1: Comparison of the execution time between different programming languages using the computer language benchmarks game [**?**]

Goal

To reduce the energy consumption of python programs, we decided to start with studying its behaviour during the most common use cases. After this we will targtet the structures of the language itself in the hope to find some generic guidelines for more generic purpose as Hasan et al. did in their paper [36]. After this we will measure the energy consumption of different python implementation for the goal of finding a non intrusive way to enhance this energy efficiency.

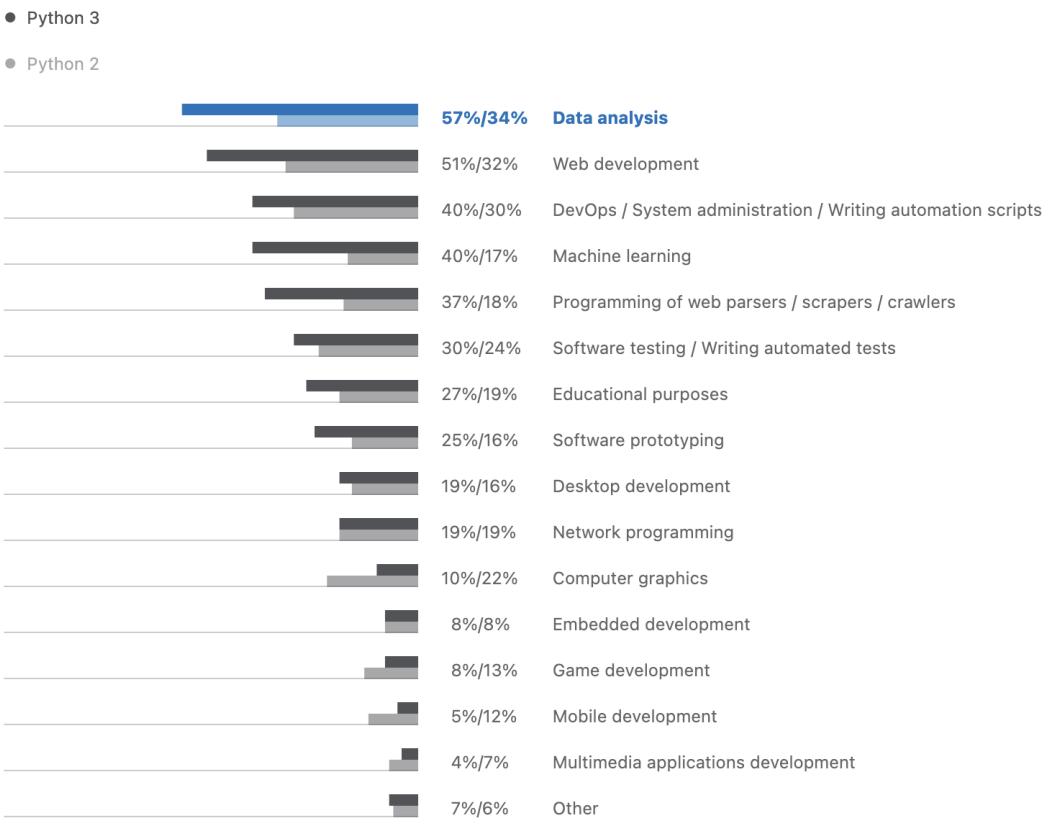


Figure 5.3: use cases of python

Energy measurement As we know, the energy of a program is the integrale of its power overtime. For us ower case, we used Intel *Running Power Average Limit* (RAPL) [44] to collect the power samples of the running tests,

We used used POWERAPI [19], to report Data collected by intel Rapl and send it into another machine that we call computing machine. then we calculate the Energy using the the trapezoidal rule.

5.4

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

Figure 5.4

figure 5.6 shows the architecture of our testing model.

The reason of separation between data collection and energy calculation is to minimise any interference with the test so our sensor is a light c program running inside a docker container.

5.3 Workload

: for this chapter we are consediring 4 seperated, yet related studies . First we are going to see the energy behavriour of python within two study cases. Web servers and Machine learning. later we will dig deeper into the energy consumption of python basic structures where finally we will conclude with the impact of python interpterer on the energy consumption. Due to the lack of of support for most of non conventional pyton interpeters, our main focus was only on micro benchmarks. Except pypy most of the python implementations don't support extra python libraries, Despite that most of the cases those extra implementations were made to optimize a specific library such as numba with numpy, or intelpython with machine learning algorthms.

preliminary studies for the first studies we used only the official version of python. because the goal was mainly to highlight the impact of the structure of code on the energy consumption. One main inconvienent of the previous methode to reduce the energy consumption, is the hard work that should be done in order to alter the existing code base in order to redcue the energy consumption. To avoid such hustle we tried to find a non intrusive way to make the python code more ecofriendly without changing its structure. One aspect of python is the fact that it is an interptered language which led many works to create their own implementation of the interpeter to enhance one or many aspect of the python code. in the following section we will discuss the impact of those implementations on the energy consumption of python programs, and in which case, one should use a non conventional interpter to save the energy consumption of their program.

To do so, we gathered a list of interpeters, transpilers and some optimization libraries. the following list, presents the most famous ones .

1. **CPython**:[5] a Python interpreter written in C programming language. It is the reference interpreter of Python. CPython compiles the source code into a bytecode and then interprets it. The CPython project still supports both versions of Python 2 and 3;

[5] https://www.python.org/[6]¸

2. **PyPy**:[7] An alternative implementation of the Python interpreter. It is written using *RPython* in order 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 will be applied only the code written by the programmer and not external libraries;

3. **Cython**:[8] a static compiler for Python. It translates the Python code into a C code, and then compiles it using a C compiler. It also support(s) an extended version of the Python language that allows programmers to call *C functions*, declare *C types* and use static types which will help the translation of Python objects into native types, such as integers, float, which often means a better performances, since native C libraries are almost all the time faster than the python written once [**?**].

4. **Intel Python**:[9] A customized interpreter developed by Intel in order to enhance performances of Python programs. It is dedicated to data sciences, high-Performance computing. It uses some Intel kernel libraries, such as Math Kernel Library (Intel MKL[10]) and data analytics acceleration library (Intel DAAL[11]). It supports both versions of Python;

5. **Active Python**:[12] developed by the Activestates company, it provides a standardized Python distribution to ensure license compliance, security, compatibility and performance. Therefore, ActivePython has its own built-in packages (more than 300 packages) and supports both versions of Python;

6. **IronPython**:[13] A .Net-based Python interpretation platform written in C# in order to be used over the .Net vm or mono, it benefits from all the optimizations of .Net virtual machine, such as the JIT and garbage collector mechanisms.

7. **GraalPython**:[14] A Python interpreter that is based on GraalVM [15] (a universal virtual machine developed by oracle for running applications written in different program-

[7] http://pypy.org
[8] https://github.com/cython/cython
[9] https://software.intel.com/en-us/distribution-for-python
[10] https://software.intel.com/en-us/mkl
[11] https://software.intel.com/en-us/intel-daal
[12] https://www.activestate.com/products/activepython/
[13] https://ironpython.net
[14] https://github.com/graalvm/graalpython/
[15] https://www.graalvm.org/docs/why-graal/

ming languages). For the time being, it only supports Python 3 and it is still in the experimental stage;

8. **Jython**:[16] Implementation of Python programming language written in Java for the *Java virtual machine* (JVM). Similar to IronPython and GraalPython, it leverages the optimization mechanisms provided by the JVM to enhance the Python programs;

9. **MicroPython**:[17] a lightweight Python version dedicated to embedded systems and micro-controllers;

10. **Nuitka**:[18] a Python compiler written in Python that generates a binary executable from Python code. It translates the Python code into a C program that is then compiled into a binary executable;

11. **Numba**:[19] a library that includes JIT compiler in order to enhance the performances of Python functions using the industry-standard LLVM compiler library;

12. **Shedskin**:[20] a static transpiler that translates implicitly statically typed python into C++ code.

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

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

15. **Stackless Python**:[21] an interpreter that focus on enhancing multi-threading programming.

16. **Pyjion**:[22] JIT API for CPython, same purpose as Parakeet and Hope.

17. **Pyston**:[23] performance-oriented Python implementation built using LLVM and modern JIT techniques. the project is funded by dropbox;

18. **Grumpy**:[24] a source-to-source transpiler that translates the Python code into a Go code that will be compiled to a binary executable. It also offers an interpreter called

[16]https://jython.github.io
[17]http://micropython.org
[18]http://nuitka.net/pages/overview.html
[19]https://numba.pydata.org
[20]https://github.com/shedskin/shedskin
[21]https://github.com/stackless-dev/stackless/wiki
[22]https://github.com/microsoft/pyjion
[23]https://blog.pyston.org
[24]https://github.com/google/grumpy

grumprun which can directly execute the Python code. Unfortunately, we cannot use it because the project is already outdated (last commit is in 2017) and it has a lot of limitation in term of supporting the Python language, such as some built-in functions and standard libraries;

19. **Psyco**:[25] a JIT compiler for Python;

20. **Unladen Swallow**:[26] an attempt to (use) LLVM as JIT compiler for CPython.

5.3.1 Runtime Classification

Before futher proceeding with the list of candidate runtimes for Python applications, we propose a classification according to several criterions:

Type refers to the category of runtime infrastructure that supports the execution of a Python application. In particular, we consider 3 types of environments: *Interpreter*, *Compiler* and *Library*. *Interpreter* refers to the class of environment that does not require any preprocessing of Python source code. *Compiler* introduces a compilation phase prior to the execution of the application. Finally, *Library* induces 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 optimisation refers to the support of *just-in-time* compilation in the runtime infrastructure supporting the execution of the application.

GC optimisation 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 should explain the classification of these runtimes

There are other implementations that we didn't considerate because either the project aborted many years ago or it has a very limited support for python features. After the collection of those implementation we filtered them. in order to keep only the versions

[25] http://psyco.sourceforge.net
[26] https://unladen-swallow.readthedocs.io/en/latest/

Table 5.2: Classification of Python implementations

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

Table 5.3: Classification of Python implementations

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

that are still maintained and support most of python features. and we classified them into 3 categories depending of their integration with the python code. In the 5.3 we describe the implementations that we kept and the version of each implementation and its category.

5.4 experimetal protocole

As we have discussed in the previous chapter 2.6. instead of running the tests, the idea was to design a system that allows practiouners to reproduce and extends our tests. and then we use the same system to run answer some of researchs question.

5.4.1 measurement context

Hardware settings all our tests have been executed in a Dell PowerEdge C6420 server machine. A summary of its hardware is listed in table 5.4. The machine is equiped with a minimal version of Debian 9 (4.9.0 kernel version) where we install Docker (version 18.09.5).

software settings For the sake of reproducibility, each experiment runs within a Docker container. for each test we create a docker image.

5.4.2 Metrics

Our focus will be mainly on CPU energy consumption because it is ten folds more than the DRAM one, since it is finit job benchmarking, time is highly correlated within the energy,

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 1
	ib0, Omni-Path, configured rate: 100 Gbps, model: Intel Omni-Path HFI Silicon 100 Series [disc

Table 5.4: Testing Machine Configuration

and it will be only useful to explain certain energytical behaviour so we wont put a lot of focus on this metric.

5.4.3 tests preparation

To study the behaviour of the python implementation regarding the energy consumption, we have to focus on the effect of the implementation and mitigate the maximum any side effects such as the organisation of the code or any extra consumption due to the operating system or tier libraries. Therefore, for each test we took the implementation written in python2 as a reference and tried to use it in other implementations as it is. If it is not supported by python3, we transformed the code using the officiel library *2to3*[27]. In the case of the libraries that use *JIT* adding a decorator to the function that we want to optimize was enough, if there are other changes we assume that they alter the original code which is against our purpose.

Each test is implemented in a Docker container for the several reasons

- Isolation: each container has only the test program implemented with a single python runtime to remove any interference between different implementations.
- Deplpoyment: to use the testing machine without extra configurations that may alter the behaviour of the os toward the enrgy consumtion
- Reproductibility: One of the most frequent benchmark crimes [**?**] in research is the lack of Reproductibility, by using Docker we ensure that each test has an Image that will be accessible in public.

Despite the presence of the official docker images for the most of the runtimes, we prefered to build our own using the same reference image in order to remove any bias due to the Os used in the offical image.We used ArchLinux with the kernel version 4.9.184 as a base image.

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

5.4.4 Extension

As we have done with the previous chapters. we provide a tool that allows to extend the tests with new workloads and new candidates. In the repository [28]. we have a tool that allows to generate new workloads and new candidates. The script `generator.py` allows to create new benchmarks by implementing a python code within different interpeters. then it generates `launcher-benchmark.sh` that can be used directly to run the expermient. Furthermore all the successful implementations are stored in seperate directory and the that couldn't work (mostly because of compatibility issues) stored in a recap file called `benchmarkTest.md`. where benchmark is the name of the new workload; To add extra **candidates** one should add a base docker file that containes the new implementation and if there should be extra manipulation that should be added to the workload files ,such as adding new decorator or changing some parameters, then they should be added as an extra function in the script `generator.py`. Finally they should be included in the python candidates.

5.5 Results and finding

web developement

first we wanted to test the energy consumption of python using on djagon,one of the most popular web frameworks. therefore we create a simple website where we try to analyse the cost of a single request using different mechanisms to fetch the data from the database. the idea behind this use case is to see wether the choice of the database and the orm (object relational mapper) impacts the energy consumption and the perfromances of the website.

we considered using two different databases postgresSQL and SQlite3 that containes the same data, and three different ways to fetch the data.

1. the naive version: which relies only on the ORM to get the data
2. Prefetch : basically we just prefetch the data before we request it
3. optimized : and here we optimize our request on the SQL level without passing by the ORM

As one can see in figure 5.7, different choices of requesting the data have a huge impact on the energy consumption. as the naive version can consume up to 10x more energy than the optimized one. In the other hand the choice of the data base didnt have huge impact on the total energy despite their different behaviour regarding the execution time and the

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

average power. This can be useful to help develeppers make a choice regarding which data base they can use based on the number of the expected requests and the expectation of the performance.

Another interesting observation is the impact of the interpreter, as figure 5.7 highligh. the use of Pypy interpter instead of the default one helped reducing the amount of the energy consumption even when we are dealing with the naive version.

Figure 5.13 shows the behaviour of python programs when we try to introduce the parallelism. As we know python is a single threaded program thx to ghe GLS (global lock system) howerver due to the increase of the number of cores /threads per cpu it many libraries stared to take advanteage of this feature so most of them they will try to simulate the multithreading by using multiple instances pf the processor

as we can see in the graph the energy consumption is correlate with the number of threads until we arrive to the limit of the cores and then we lose the adventage of the multiprocessing , well in that case we pass from parallelism to concurency. where different sub process have to compete for the CPU Ressources. Another finding is when we hit the number of physical cores. there was an increase of execution time but still reduced energy, the reason behind this is the scheduler of the operating system. basicaly he favorits the hyper threads than the physical core wich will lead to some context switches which cause the slow behaviour, howerver in the other hand the other two physical cores are not consuming energy, neither their hyper threads which explains the gain of the eneryg consumption in this case. in the Chapter we discuss a deeper this behaviour of the scheduler and in that case we confirm that it is not related to python but it is more generic behaviour

python and multiprocessing

Figure 5.13 shows the behaviour of python programs when we try to introduce the parallelism. As we know python is a single threaded program thx to ghe GLS (global lock system) howerver due to the increase of the number of cores /threads per cpu it many libraries stared to take advanteage of this feature so most of them they will try to simulate the multithreading by using multiple instances pf the processor

as we can see in the graph the energy consumption is correlate with the number of threads until we arrive to the limit of the cores and then we lose the adventage of the multiprocessing , well in that case we pass from parallelism to concurency. where different sub process have to compete for the CPU Ressources. Another finding is when we hit the number of physical cores. there was an increase of execution time but still reduced energy, the reason behind this is the scheduler of the operating system. basicaly he favorits the hyper threads than the physical core wich will lead to some context switches which cause the slow behaviour,

howerver in the other hand the other two physical cores are not consuming energy, neither their hyper threads which explains the gain of the eneryg consumption in this case. in the Chapter we discuss a deeper this behaviour of the scheduler and in that case we confirm that it is not related to python but it is more generic behaviour

5.5.1 python insights

The goal of this section is to find some insights where we can optimize the energy consumption without impacting the source code. so we started by extending the work of **?** and **(author?)** to the python environement.

another field of investigation is the type data and structure conroles. that might impact the energy consumption. To do so. We iterate over a list using three methodes. first the classical for (for i in range (len(n))). howerver as we can see here unlike other porgramming langauges it requires extra operations such as determining the length of the collection , and then using the iterator range. so we tried the more adapted version (for element in collection). Morever in most programming languages the for loop is translated to a while loop (transfirmation from D type to B type – asm –), therefore we wanted to compare this with a (while) version. After determining the main ways to iterate over a loop we run the algorithms collections of different data types to see wether it will impact of the energy consumption of the code, same thing for the size of the collection.

As we can see in figure 5.10 the type of the data hasn't an impact on the energy consumption.In the other hand the way we iterate over the collection has a huge factor. interestely, the for in range loop was by far the optimal one with following by the regural for in collection, and the while part was the least one with an overhead of 400% compared to the first option.

the reason behind a such behaviour is mainly related on how python interpeter is made. In order to reduce the latency of the python code. most of the buitl-in functions an operations are written in C, same thing goes for the function range, further more the function len has a complexity of O(1) because it is based on the function Py_SIZE of C which stores the length in a field for the object.

so basically for the for in range is basically creating a new iterator that has the same length of the first one. and for each iteration we will have to do a second acceess (l[i]) instead of one one access hence the double time.

for the wile is even slower due to the implicite incrementation of the variable which will cause an extra operation during the loop .

to confirm the hyphothsis. we tried to construct a new list by editing the elements of the previous one. And as predicted , the builtin methodes are the best energy saving, while the customize while loop is the heaviest. an interesting finding is. the impact of annonymous

functions (lambda expressions) on the energy consumption. the reseason behind this is the fact that python treats those functions as local variables unlike the predifined ones which are global in our case. therefore they are faster and consumes less energy

conclusion This study has shown us, that the optimial way to reduce the energy consumption of the python code is to follow the guidelines and use the builtin functions, which is kinda happy news for the developers since they don't have to make extra effort to make their code green

5.5.2 python runtimes

As we can see in figure 5.15, there is no evolution between cpython2 and cpython3 Intelpython and active python both folows the same behaviour. one can conclude that the work that have been on those interpreters is mainly to improve a specific purpose, Active python claims that their version is focused on secuirty which explains the lack of some performances due to the introduction of more reonforcment .Morver Intel published their version of python as a dedicated for machine learning. Unfortunately the tommti benchmark is a set that focuses on the general purpose programming which does not reflect the performance of the machine learning. Another aspect of a such behaviour might be due to the fact of the processors that were used in the testbed. and it might change in the future if we include the GPU part for the machine learning benchmarks As for nuitka. there were no optimization in the energy consumpotion despite the fact that is it a complier. However if we dig through the nuitka mechanisms. they basically embed the python code with an interpreter. Unllike nuitka shedskin exhebit a the best energy consumption pattern when it comes to the arithmetic operartions. One can conclude it is due to the fact of the native type of the variables. unlike the interpreters where they are treated as object in the begining.

for the other interpeters pypy is very promising especially when it comes to data manipulation as one can see in the figure **??** pypy is by far the best interperer when it comes to treating vectors. numba2 introduced the JIT but wasn't as promising as numba3.

for the other vm based interpereters. jython and ipy lacked in term of energy optimisation which was kinda expected since they were in their the begining of the stage and the main puprose of such implementation is to link the bytecode generated by jython and ironpython with their respectives virtual machin.

Unlike the the previous interpeters. graal exhebits a certin promises when it comes to complex algorithms - nested loops - micro python is dedicated to embeded systems so lunching it is powerful cpu machines will be misleading.

Most of the interperters had the same behaviour when it comes to the input outputs. except for jython which was kinda of a anomaly probably due to the lack of the optimizations.

5.6 Threads to validity

5.7 tools and contributions

5.8 conclusion

one can notice that the choice of the right interpeter can impact havely the energy consumption of the python programs. The fact that there is no general solution make this study more interesting. however the main drawback is the lack of the compatibility for some of those solutions which lead to some sacrifices when we want something generic

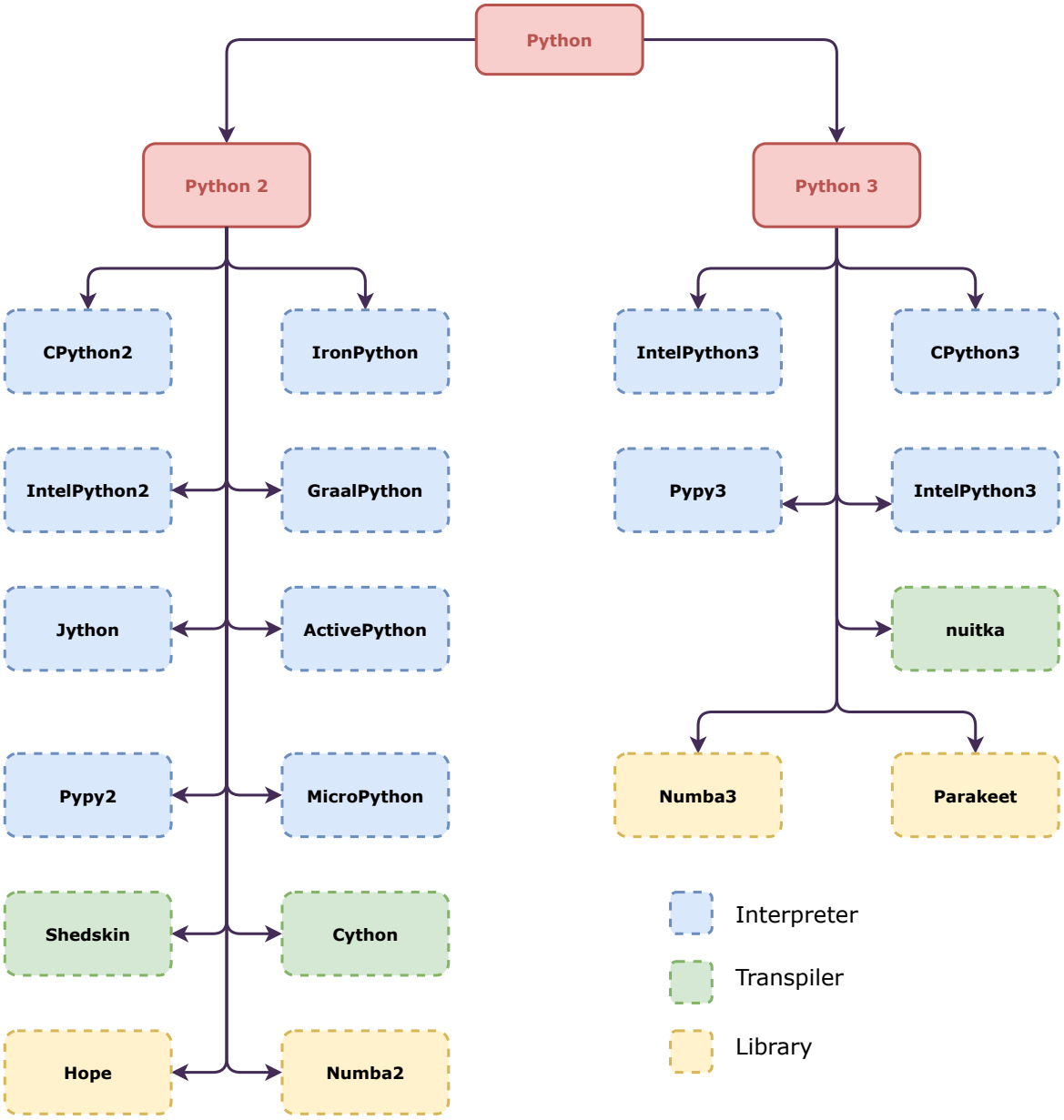


Figure 5.5: Python interpeters

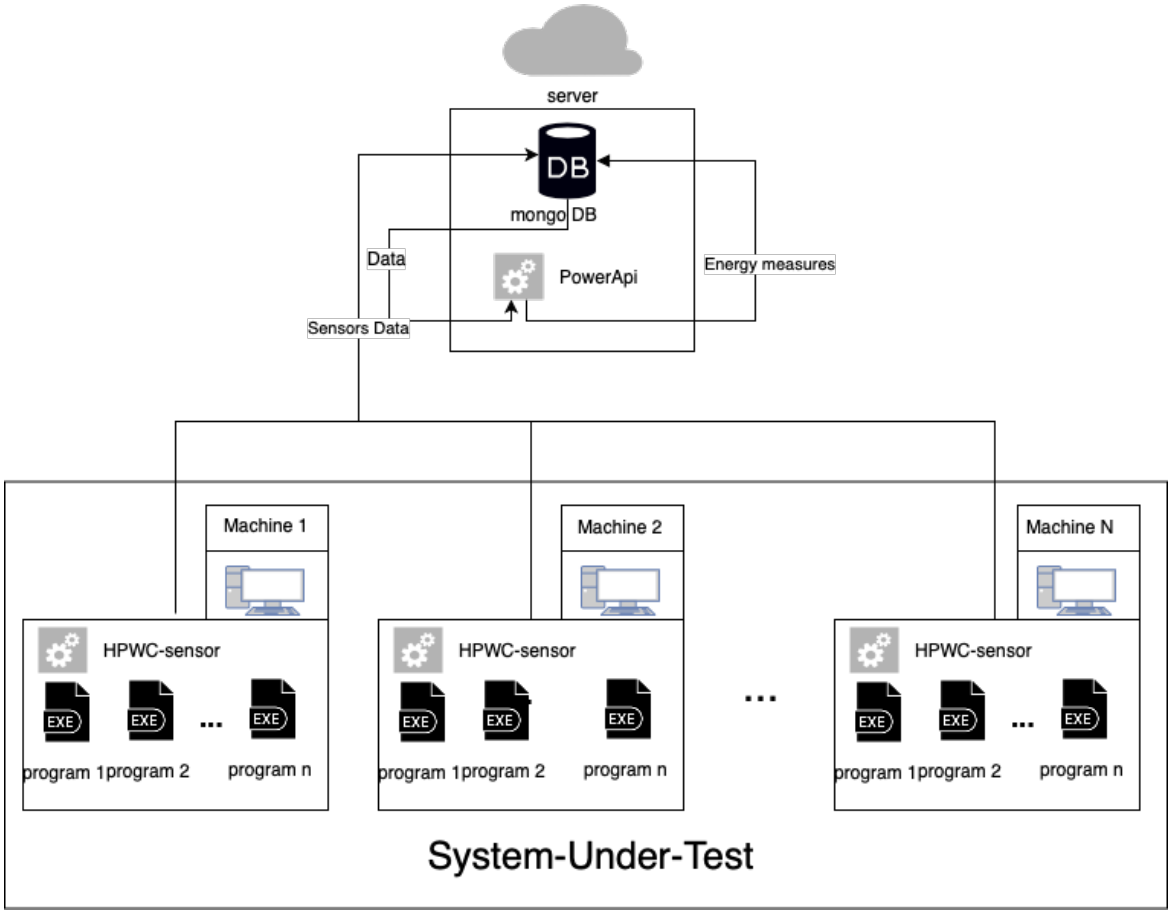


Figure 5.6: powerapi architecture

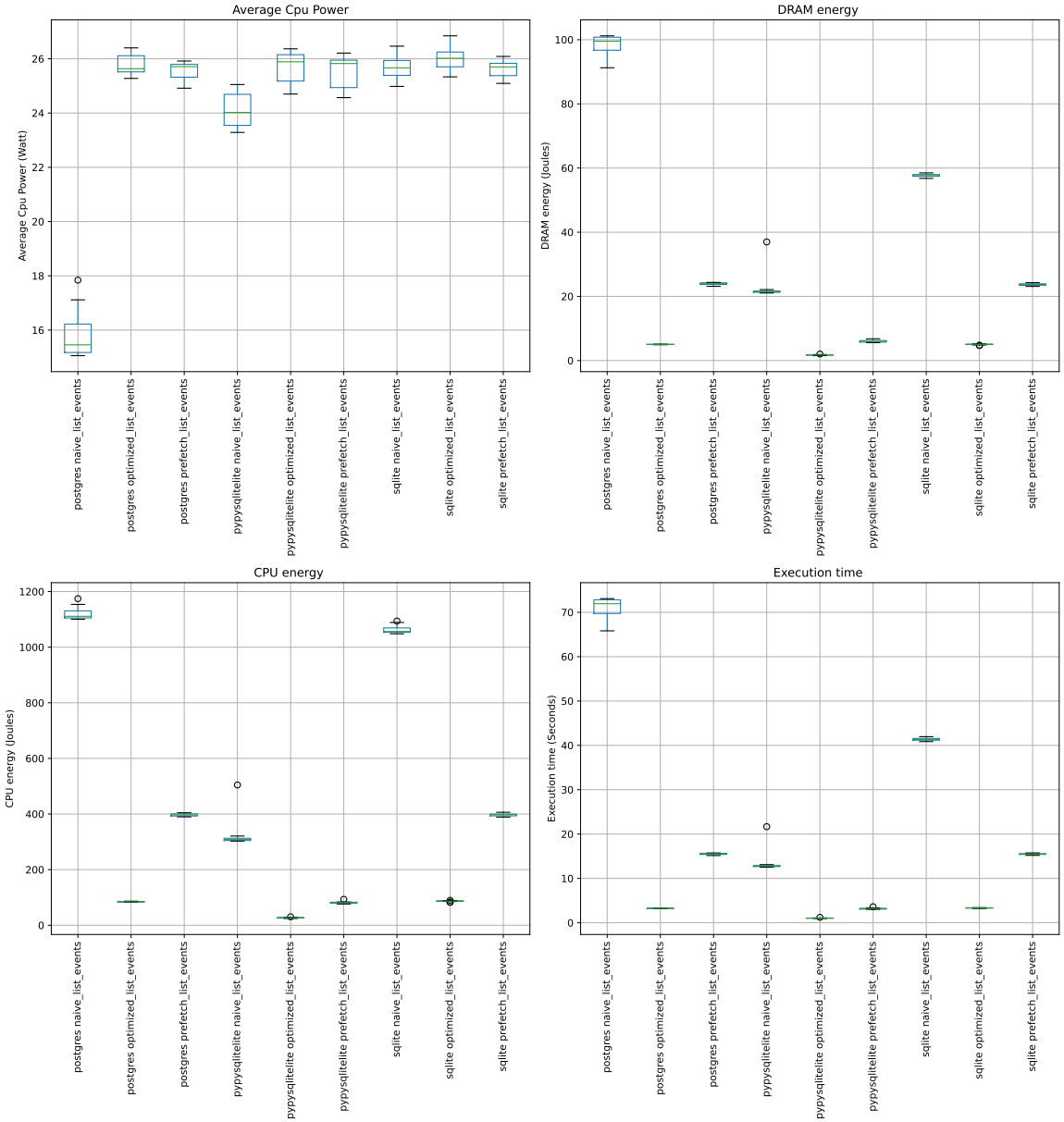


Figure 5.7: energy behaviour based on multiprocessing

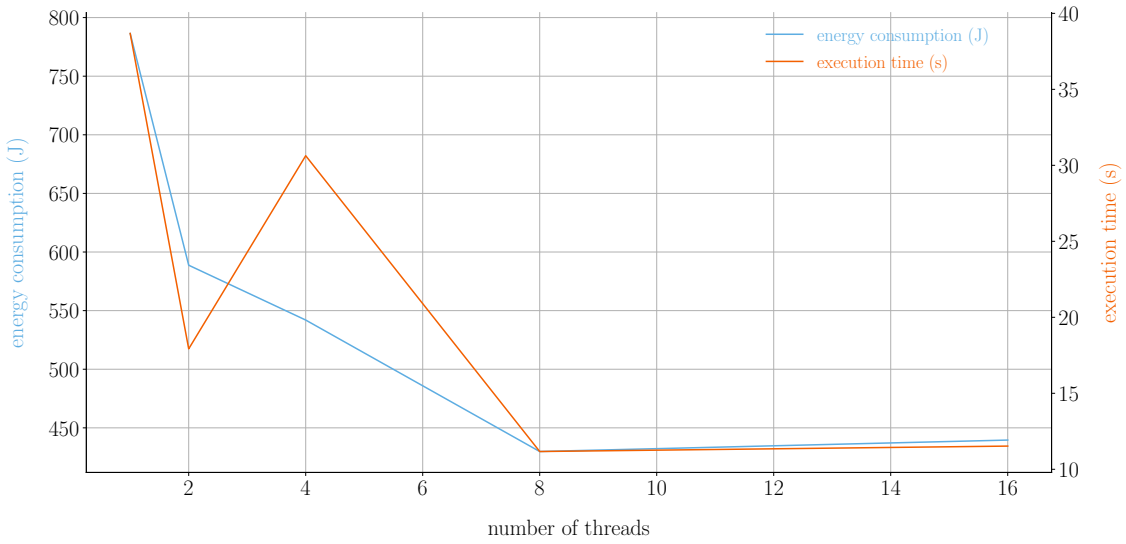


Figure 5.8: energy behaviour based on multiprocessing

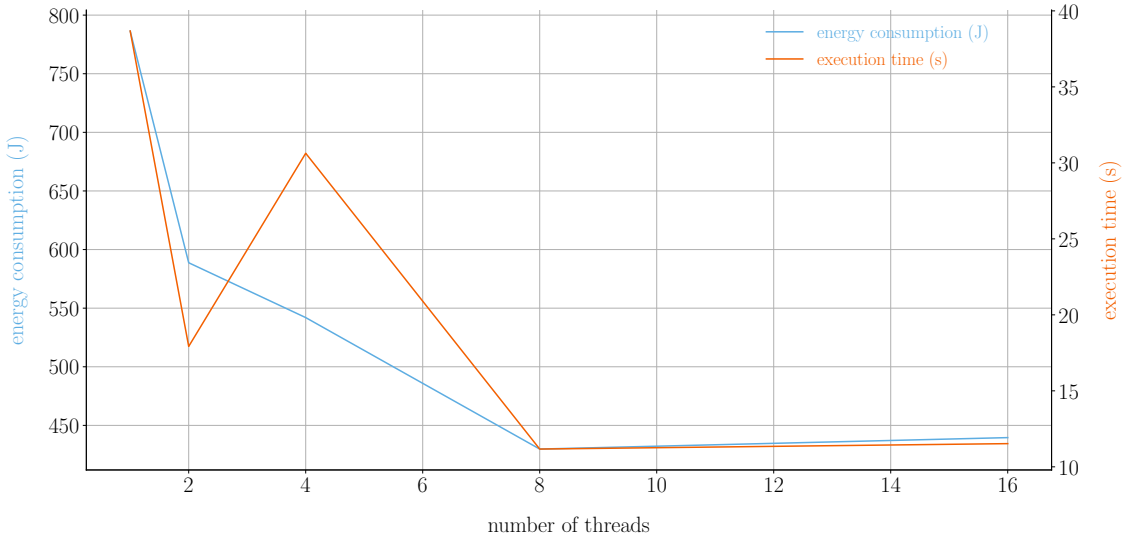


Figure 5.9: energy behaviour based on multiprocessing

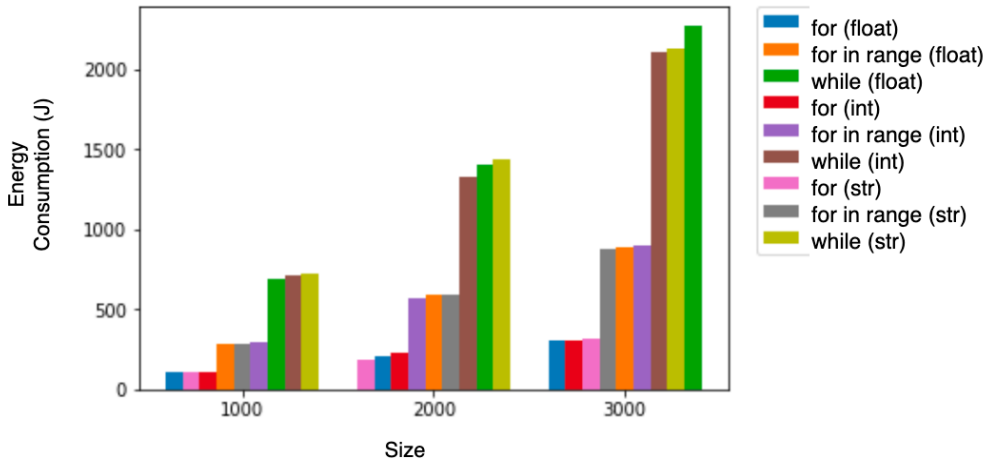


Figure 5.10: energy behaviour of different python loops

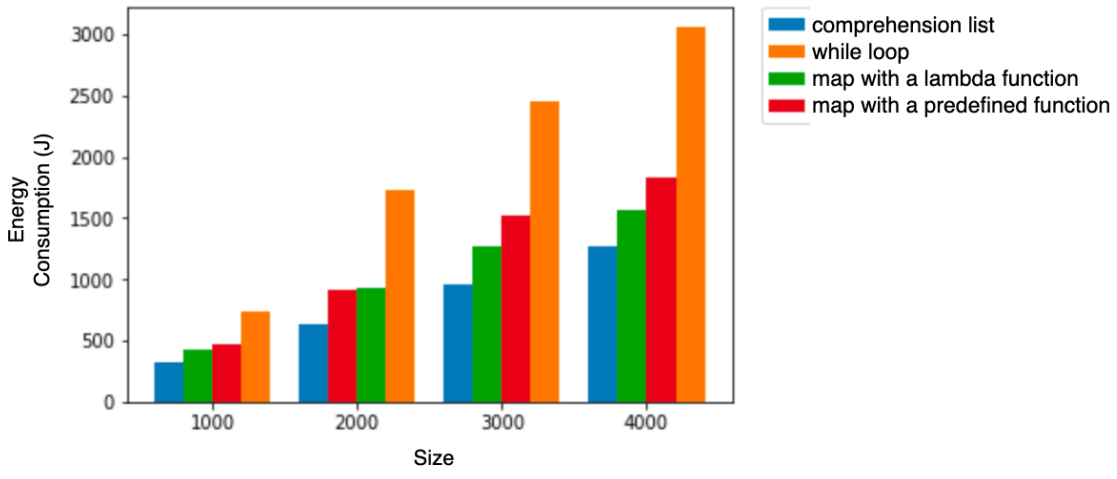


Figure 5.11: energy behaviour of different methodes to change a list

Figure 5.12: energy consumption of different impelementations using Bit Operation benchmarks (Joule)

benchmark	A	GG	LL	Or	XOR
activepython	676.980672	763.208040	651.783048	743.016425	728.828481
cpython2	441.082298	435.886950	430.846171	415.247383	419.081447
cpython3	595.209019	685.085315	563.839300	657.972734	655.560574
cython	35.077408	182.688395	274.177830	34.868014	34.504778
nuitka	33.260991	32.980380	33.256450	33.472770	33.030889
numba2	9.102939	8.411176	9.460620	9.375920	9.755952
numba3	9.566646	10.144212	9.219703	9.344613	9.665108
pypy2	8.456867	7.844918	8.286278	8.138692	7.952999
pypy3	7.552731	8.093884	8.108508	8.669448	8.623737
shedskin	8.024198	8.070870	8.399917	8.126236	8.277546

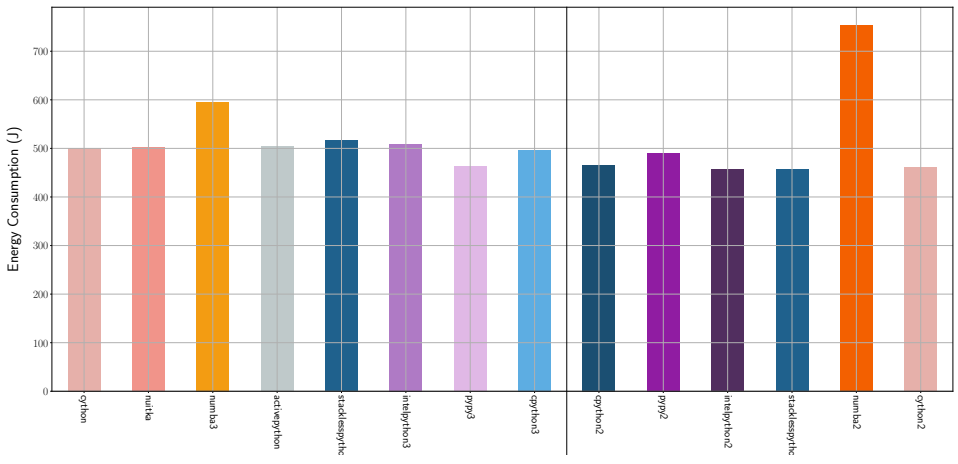


Figure 5.13: energy behaviour based on multiprocessing

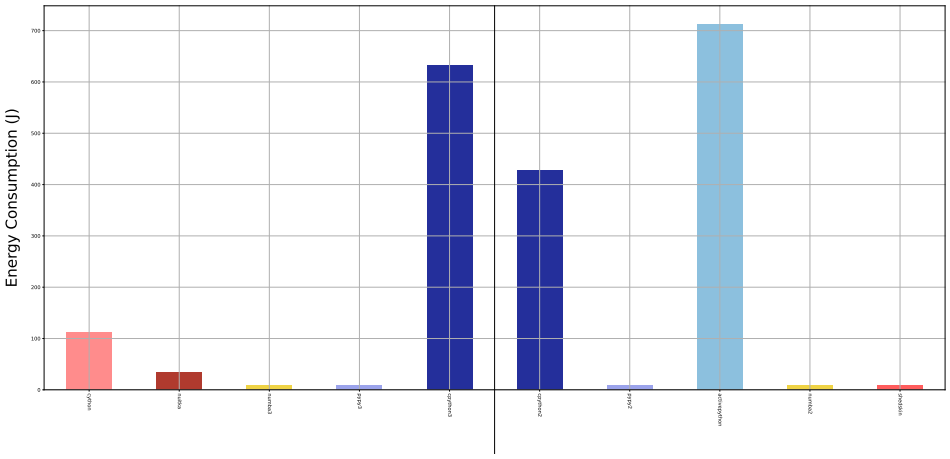


Figure 5.14: Mean consumption of different implementations of bit operations (Joule)


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

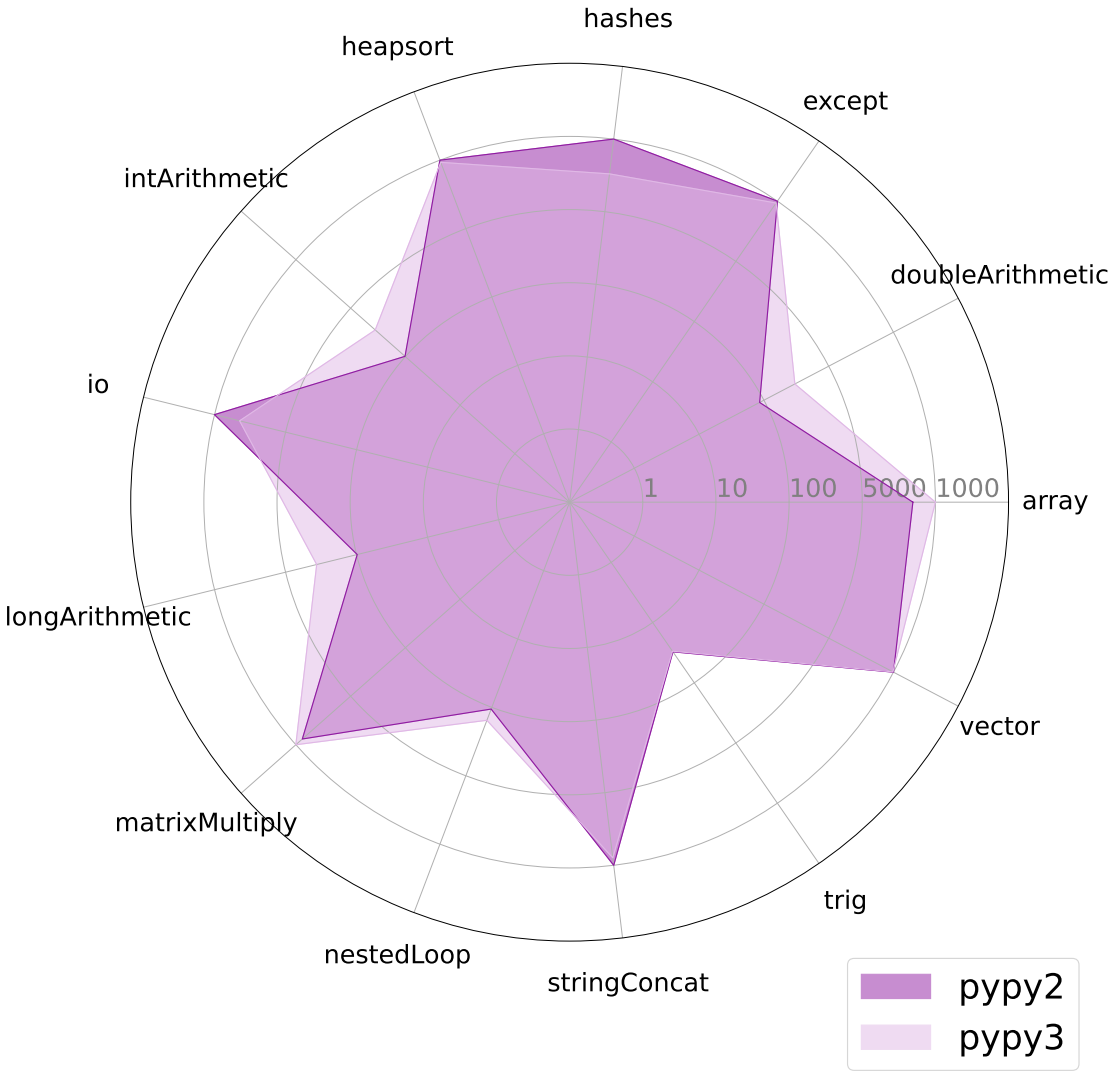


Figure 5.16: green factor of pypy

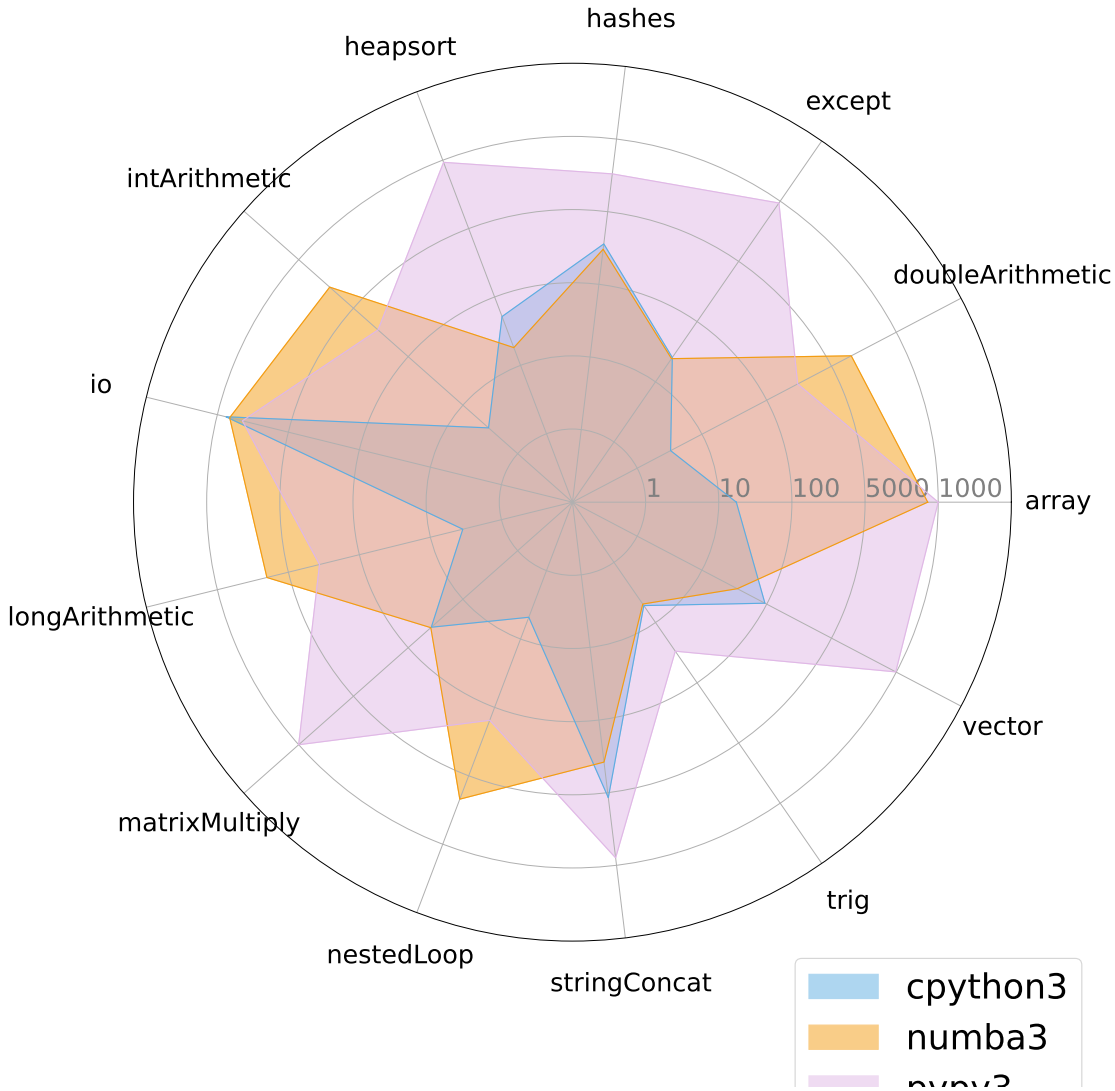


Figure 5.17: comparaison of pypy vs python vs numba

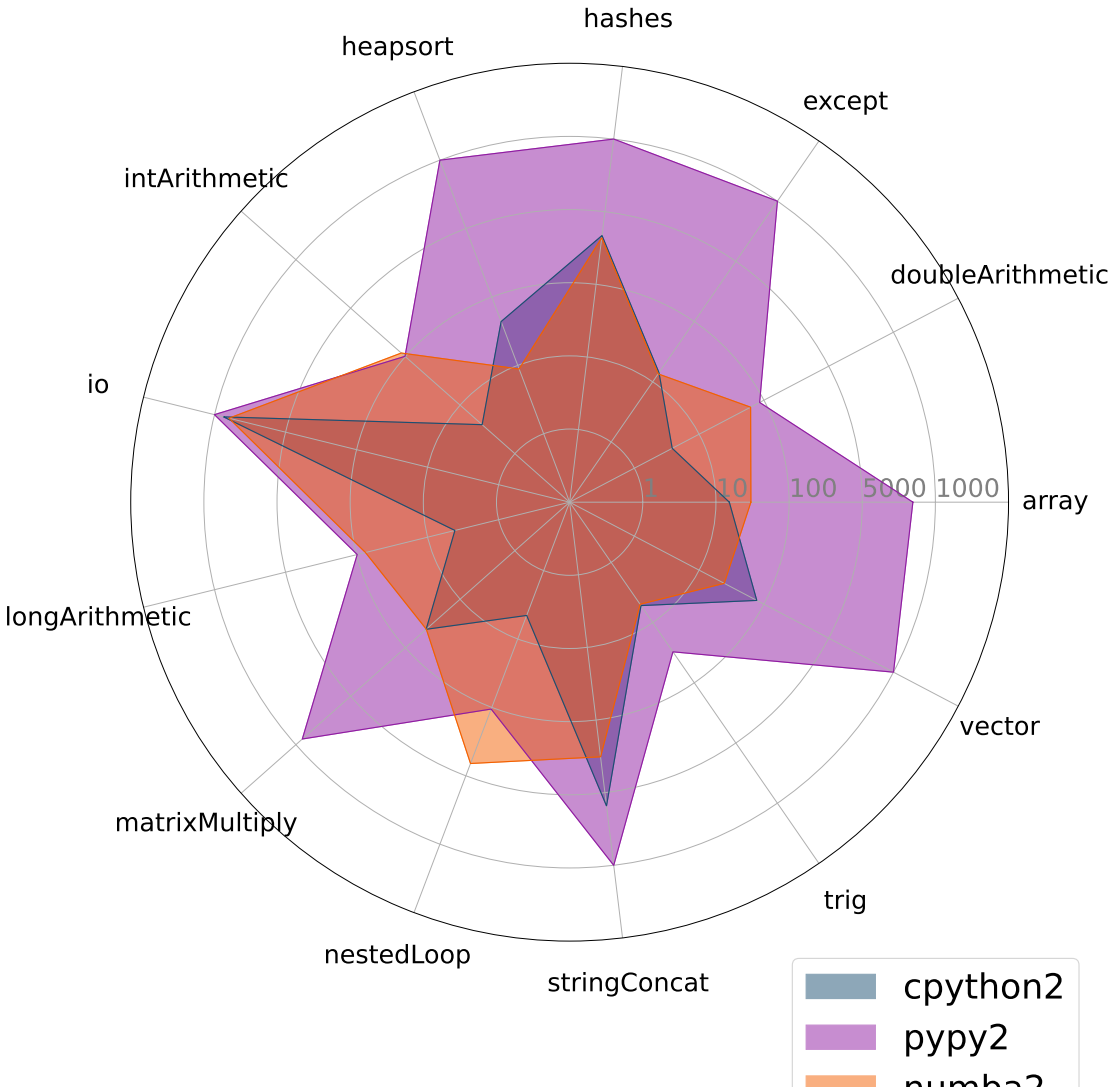


Figure 5.18: comparaison of pypy vs python vs numba

Chapter 6

The impact of Java virtual machine on Java programs

JAVA

6.1 Introduction

As reported in the state of the art, Java is one of the most popular programming languages adopted by practitioners.Furthermore, if we take into consideration legacy applications, Java becomes the most used programming langues.In addition to its popularity, Java exhibits an interesting behaviour when it comes to energy consumption and performances, basically Java applications can be at the same time one of the most energy efficient or hungry solution. 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. Therefore, we wanted 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 the energy consumption.

6.1.1 Goal

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 question we will run an empirical study to highlight the impact of this runtime.

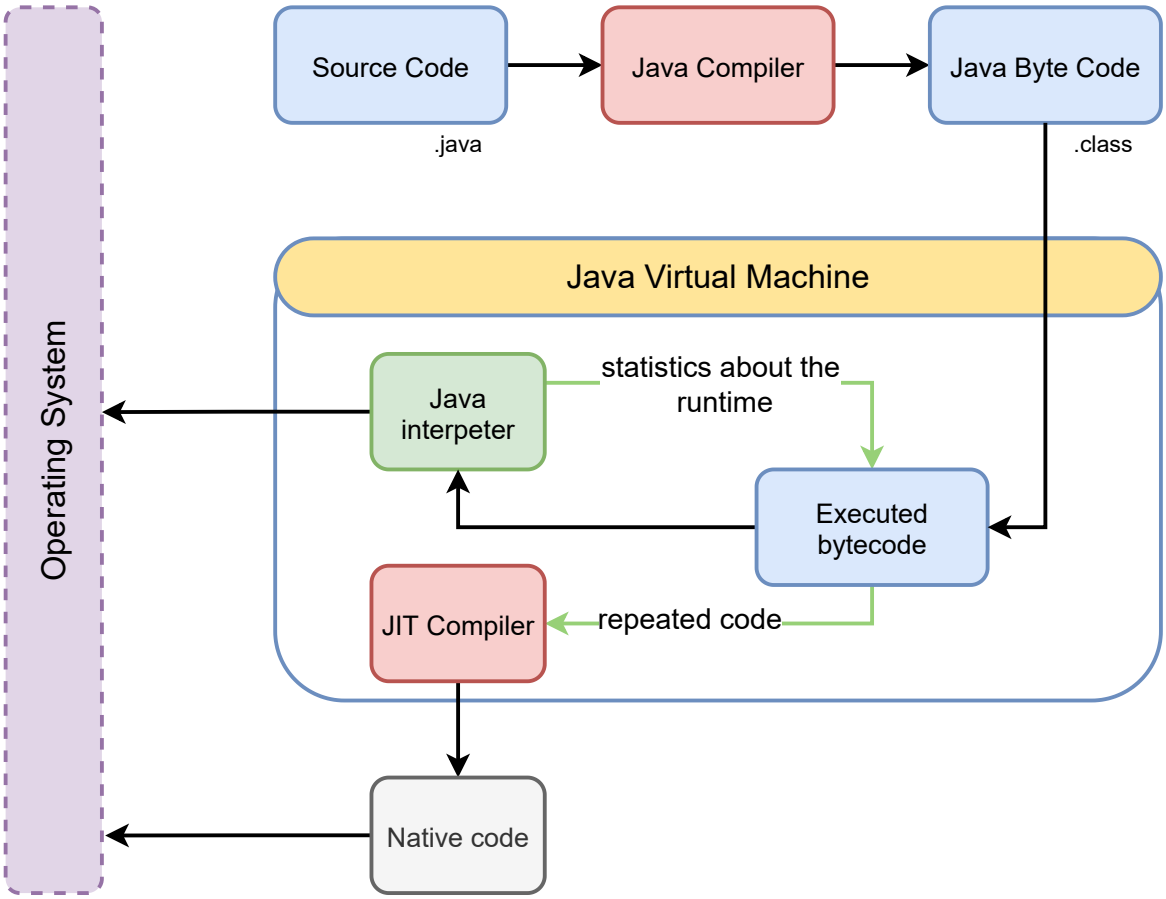


Figure 6.1: JVM architecture

6.1.2 definition of the JVM

6.2 Experimental Protocol

To investigate the effect that could 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 using several established Java benchmarks and JVM configurations.

6.2.1 Measurement Contexts

Software Settings. For the sake of reproducibility, each experiment runs within a Docker container based on SDKMAN image and Alpline docker.

Hardware Settings. To report on reproducible measurements, we used the cluster Dahu from the G5K platform [5] 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

Table 6.1: List of selected JVM distributions.

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

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 to conduct robust experiments and reduce the factors that can affect the energy consumption measurements during our experiments [68].

Java Virtual Machines Candidates. We considered a set of 52 JVM distributions taken from 8 different providers/packagers mostly obtained from SDKMAN!,[1] as listed in Table 6.1. Depending on providers, either all the versions, majors, or LTS are made available by SDKMAN!.

6.2.2 Workload

We ran our experiments across 12 Java benchmarks we picked from OpenBenchmarking.org.[2] This includes 5 acknowledged benchmarks from the DACAPO benchmark suite v. 9.12 [10], 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 communities [50, 43]. 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 [**?** 72], namely ALS, Dotty, Fj-kmeans, Neo4j, Philosophers, Reaction and Scrabble, which offers a diversified set of benchmarks aimed at testing JIT, GC, profilers, analyzers, and other tools. The benchmarks we picked from both suites exercise a broad range of programming paradigms, including concurrent, parallel, functional, and object-oriented programming. Table 6.2 summarizes the selected benchmarks with a short description.

Fighre **??** highligts the scope of each benchmark from the test suit.

[1] https://sdkman.io/
[2] https://openbenchmarking.org

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

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

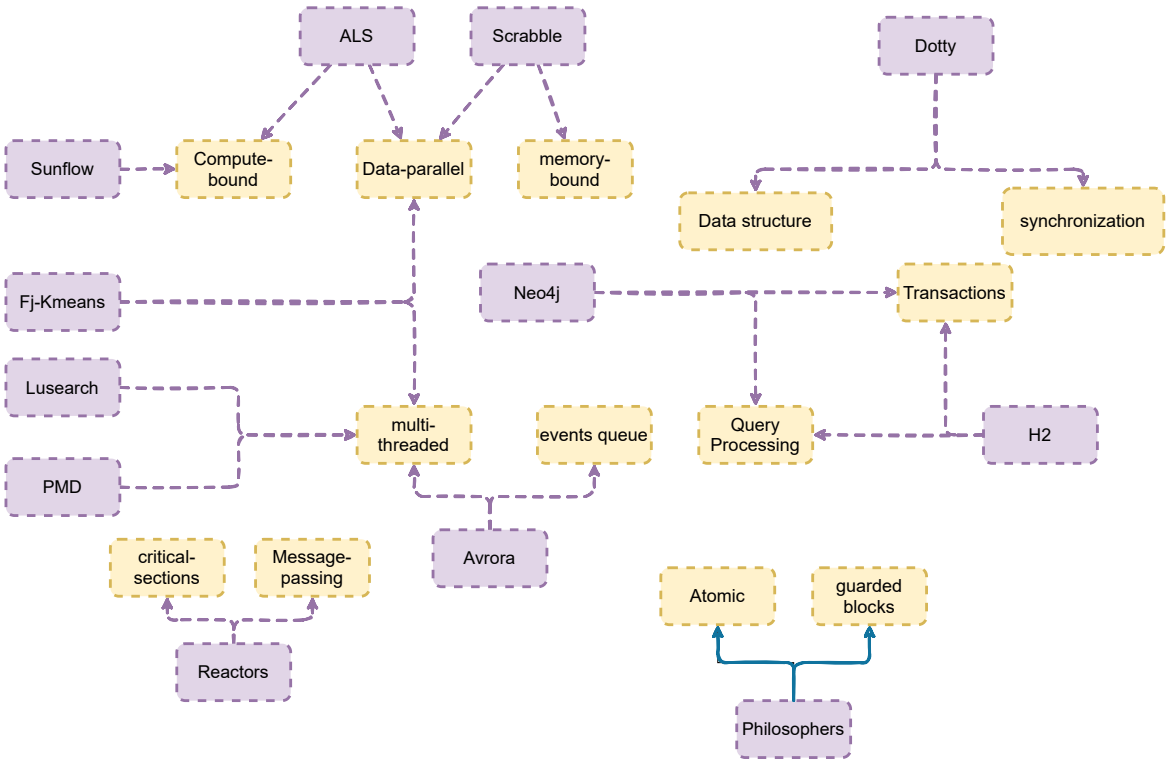


Figure 6.2: Target scope of Dacapo and Renaissance benchmarks

6.2.3 Metrics and measurement

Since the goal of this study is the green aspect Java virtual machines, our key metric will be the energy consumption of the job completed by each configuration. In addition to the energy consumption, we collected additional metrics to explain the reasons behind the behaviour of each experiment. Those extra 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 [44, 23]. We note that, due to CPU energy consumption variations issues [68], we used the same node for all our experiments. Moreover, we tried to be very careful, while running our experiments, not to fall in 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.[3]

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

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

6.2.4 Extension

We also added an extension to the protocol to allow the user to run the same experiment with different configurations. The package is available in the github repository [4]. To add extra **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. ***TODO : The jvm must be provided by sdkman***[5] For the **workload**,the benchmarks should be provided in the `benchmarks` directory. As for extra **metrics** One can create a new script file that monitor the experiment and record the metrics. We provide some examples such as `recordpower.sh` to measure the instantanious power and `recordthreads.s` to measure the number of active threads during the experiment.

For faster experements we propose **Jreferral**[6] an open source that provides an opensource to compare the energy consumption of differents jvms for any java application. Later we will discuss this tool in the section 7.2

6.3 Experiments & Results

6.3.1 Energy Impact of JVM Distributions

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

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

[4]https://github.com/chakib-belgaid/jvm-comparaison
[5]https://sdkman.io/
[6]https://github.com/chakib-belgaid/jreferral

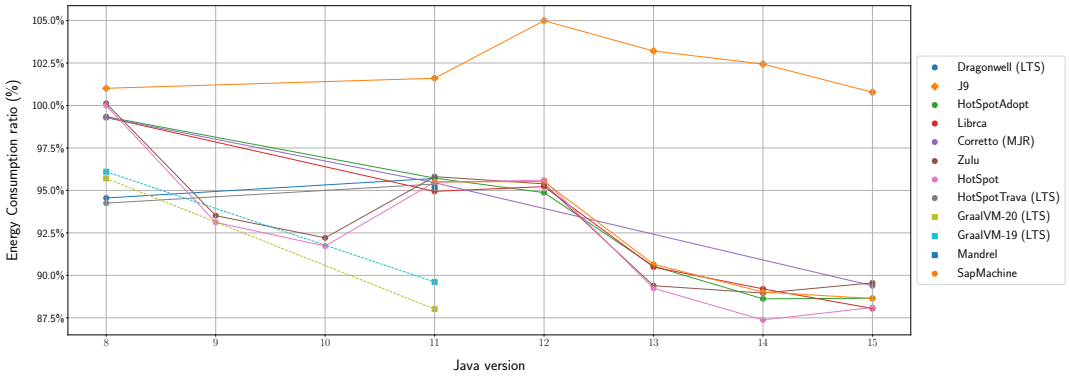


Figure 6.3: Energy consumption evolution of selected JVM distributions along versions.

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 that its recent releases provide the best energy efficiency for Java 11, but recent releases of other distributions seem to reach similar efficiency for Java 13 and above, which are recent versions not supported by GRAALVM yet.

Interestingly, this convergence of distributions has been observed since Java 11 and coincides with the adoption of DCE VM by HOTSPOT. Ultimately, 3 clusters of JVMs that encompass JVMs with similar energy consumption can be seen through Figure 6.3: J9, the HOTSPOT and its variants, and GRAALVM. Additional detailed figures to illustrate the evolution of energy consumption per benchmark/JVM are made available from the online repository.[7]

Then, Figure **??** depicts the evolution of the energy consumption of the 12 benchmarks, when executed on the HOTSPOT JVM. Figure 6.4 reports on the energy consumption variation of individual benchmarks, using to HOTSPOT-8 as the baseline. Our results show that the JVM version can severely impact the energy consumption of the application. However, unlike Figure 6.3, one can observe that, depending on applications, 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

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

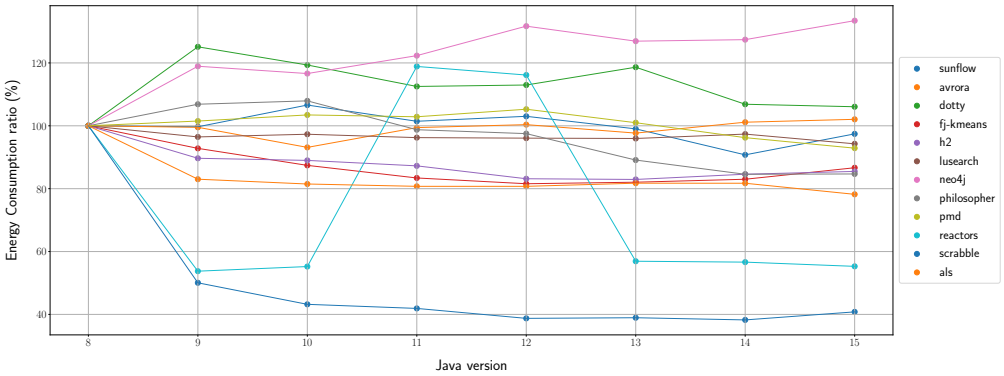


Figure 6.4: Energy consumption of the HotSpot JVM along versions.

of `VarHandle` to allow low-level access to the memory order modes available in JDK 9 and work along `Unsafe Classe` that was removed from from JVM 11.[8]

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

Figure 6.5 further explores the comparison of energy efficiency of the JVM distributions per benchmark. One can observe that, depending on the benchmark's focus, the energy efficiency of JVM distributions may strongly vary. When considering individual benchmarks, J9 performs the worst for at least 6 out of 12 benchmarks—*i.e.*, 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 a low energy consumption for all benchmarks, except for Reactors and Avrora. Yet, some differences still can be observed with HOTSPOT depending on applications. The newer version of HOTSPOT-15 was averagely good and, compared to HOTSPOT-8, it significantly enhances energy consumption for most scenarios. Finally, Neo4J is the only selected benchmark where HOTSPOT-8 is more energy efficient than HOTSPOT-15.

Service-oriented applications. In this section, instead of considering bounded execution of benchmarks, we run the same benchmarks as services for 20 minutes, and we compare the average power and total requests processed by each of the 3 JVM distributions. Globally, the

[8]https://blogs.oracle.com/javamagazine/the-unsafe-class-unsafe-at-any-speed

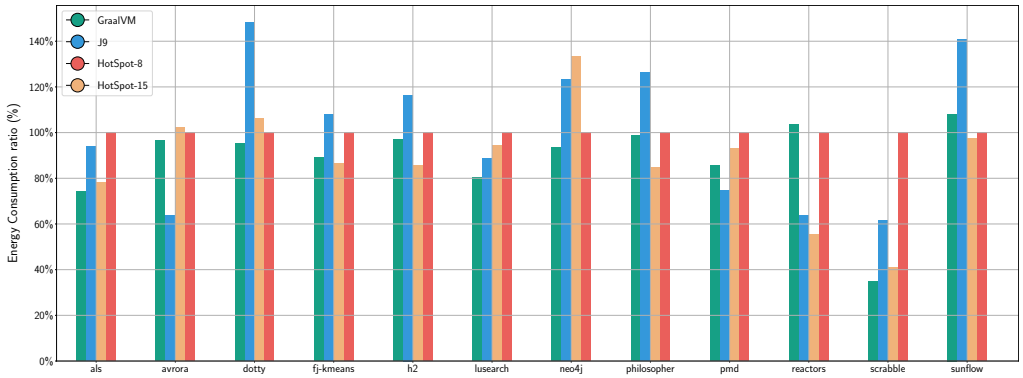


Figure 6.5: Energy consumption comparison across Java benchmarks for HOTSPOT, GRAALVM & J9.

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

Benchmark	**JVM**	**Power (P)**	**Requests (R)**	$P/R \times 10^{-3}$
Scrabble	GRAALVM	109 W	**5**,336 req	**2**0 mW
	HOTSPOT	98 W	3,595 req	27 mW
	J9	**9**2 W	2,603 req	35 mW
Dotty	GRAALVM	**4**5 W	510 req	88 mW
	HOTSPOT	**4**5 W	**5**97 req	**7**5 mW
	J9	46 W	381 req	120 mW

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

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.

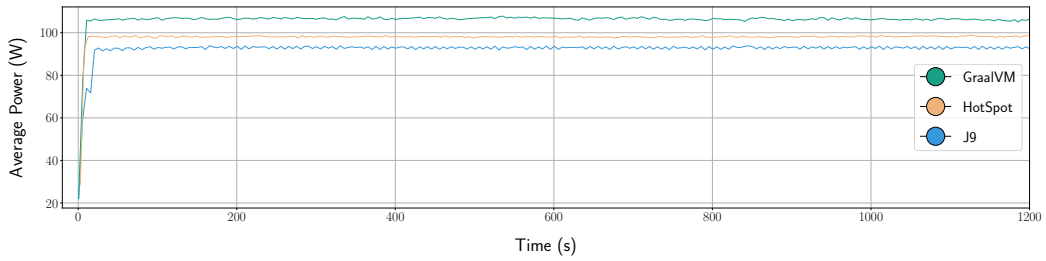


Figure 6.6: Power consumption of Scrabble as a service for HOTSPOT, GRAALVM & J9.

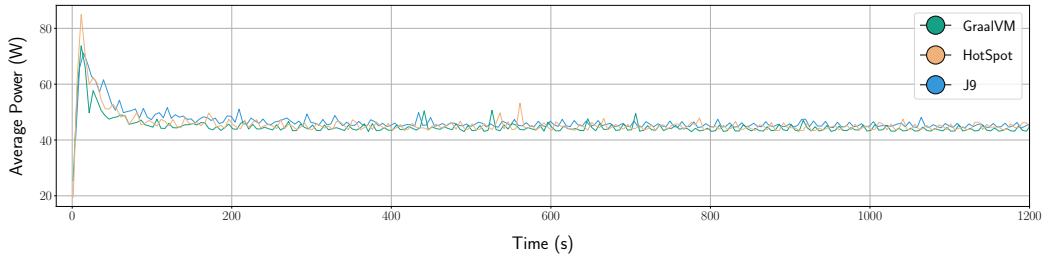


Figure 6.7: Power consumption of Dotty as a service for HOTSPOT, GRAALVM & J9.

At the beginning of the execution, GRAALVM has a slightly lower power consumption, is faster, and consumes 10% less energy. After about 150 seconds, the power differences between the 3 JVMs is barely noticeable. One can, however, notice the effect of the JIT, as HOTSPOT takes the advantage over GRAALVM and becomes more energy efficient. In total, HOTSPOT completes 597 requests against 510 for GRAALVM and 381 for J9, as shown in Table 6.3. HOTSPOT was thus the best choice on 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 exactly what we did in our experiments, and why HOTSPOT was more energy efficient than GRAALVM in Figure 6.5, thus ignoring the warm-up phase would have 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, that strongly depends on the application context. When it comes to the JVM version, latest releases tend to offer the lowest power consumption, but experimental features should be carefully configured, thus further questioning the impact of JVM parameters.

6.3.2 Energy Impact of JVM Settings

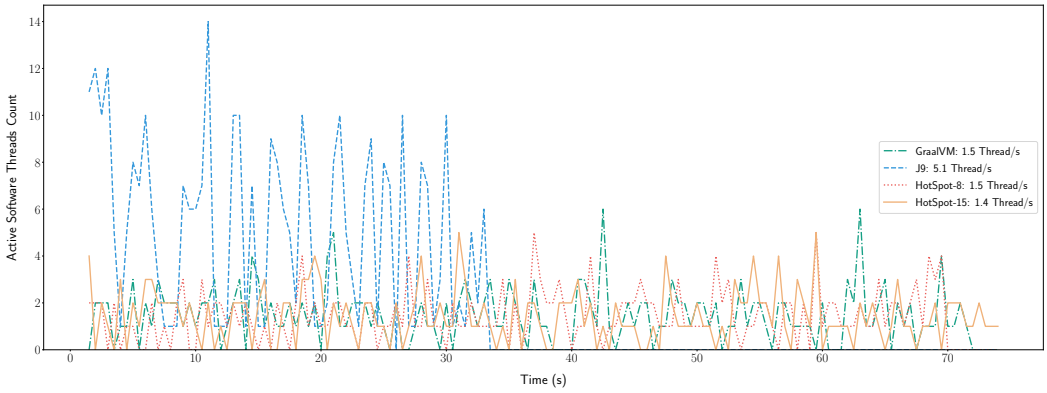
The purpose of our study is not only to investigate the impact of the JVM platform on the energy consumption, but also the different JVM parameters and configurations that might have a positive or negative effect, with a focus on 3 available settings: multi-threading, JIT, and GC.

Multithreading

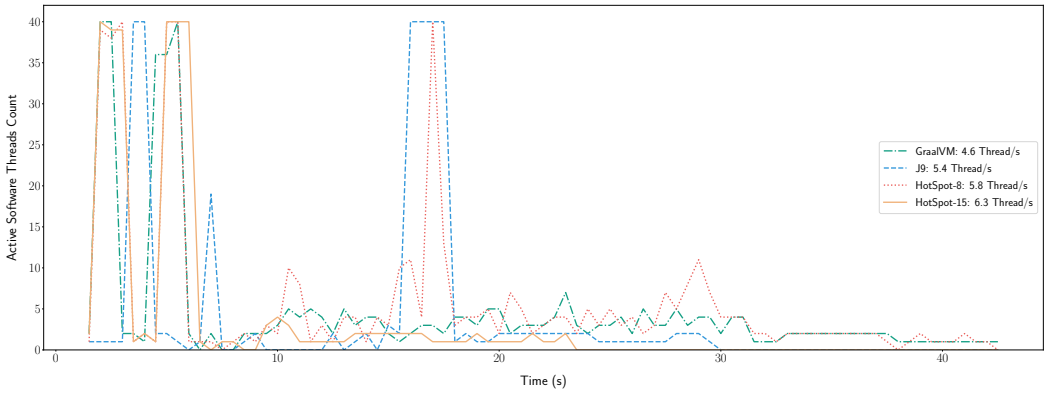
The purpose behind this phase is to investigate the impact JVM thread management strategies on the energy consumption. This encompasses exploring if the management strategies of application-level parallelism (so called *threads*) results in different energy efficiencies, depending on JVM distributions.

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

Figure 6.8 delivers a closer look to the thread allocation strategies adopted by JVM. First, Figure 6.8a illustrates the active threads count evolution over time (excluding the JVM-related threads, usually 1 or 2 extra threads depending on the execution phase) for Avrora. One can notice through the figure that J9 exploits the CPU more intensively by running much more parallel threads compared to other JVMs (an average of 5.1 threads per second for J9 while the other JVMs do not exceed 1.5 thread per second). Furthermore, the number of context switches is twice bigger for J9, while the number of soft page faults is twice smaller. The efficient J9 thread management explains why running the Avrora benchmark took much less time and consumed less energy, given that no other difference for the JIT or GC configuration was spotted between the JVMs. Another key reasons of 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-collectable *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 own heap and no deadlock can occur.



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



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

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

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

In conclusion, JVMs thread management can sometimes constitute a key factor that impacts software energy consumption. However, we suggest to check and compare 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).[9] 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 complication (that generates compiled versions of methods that collect profiling information about themselves), and we also varied the JIT maximum compilation level from 0 to 4, we also tried out HOTSPOT with a basic GRAALVM JIT. We note that the 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 amount of extra tuning. The JIT C2 (also called server-side JIT) compiler only kicks-in at level 4.

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

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

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

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

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

The *p*-values are computed with the mann-whitney test, with a null hypothesis of the energy consumption being equal to the default configuration. The *p*-values in bold show the values that are significantly different from the default configuration with a 95% confidence, where the values in green highlight the strategies that consumed significantly less energy than 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 the 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. In fact, the usage of the C2 JIT is often beneficial (JIT level 4) in most cases, while using the GRAALVM JIT reported similar energy efficiency. Yet, some benchmarks showed that using only the C1 JIT (JIT level 1) is more efficient and even outperforms the usage of the C2 compiler. 10% on Avrora and 30% on Pmd are examples of energy savings observed by using the C1 compiler. However, being limited to the C1 compiler can also cause a huge degradation in energy consumption, such as 32% and 34% of additional energy consumed for the Dotty and FJ-kmeans benchmarks, respectively. Hence, if it is a matter of not using the C2 JIT, the experiments have shown that the level 1 JIT is always the best, compared to levels 2 or 3 that also use the C1 JIT, but with more options, such as code profiling that impacts negatively the performance and the energy efficiency. Level 0 JIT compilation should never be an option to consider. No *p*-value has been computed for Level 0, due to the limited amount of iterations executed with this mode (very high execution time, clearly much more consumed energy).

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

Table 6.5: The different J9 GC policies

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

Table 6.6: The different HOTSPOT/GRAALVM GC policies

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

Garbage Collection

Changing or tuning the GC strategy has been acknowledged to impact the JVM performances [51]. To investigate if this impact also benefits to energy consumption, we conducted a set of experiments on the selected JVMs. We considered different garbage collector strategies with a limited memory quantity of 2 GB, and recorded the execution time and the energy consumption. The tested GC strategies options mainly vary between J9 and the other 2 JVMs, as detailed in Table 6.5.

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

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

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

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

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

in bold show the values that are significantly different from the default configuration, where the values in green highlight the strategies that consumed significantly less energy than 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 twice energy for the Neo4j benchmark and 28% more energy for Reactors. The reason is that Metronome occurs in small preemptible steps to reduce the GC cycles composed of many GC quanta. This suits well for real-time applications and can be very beneficial when long GC pauses are not desired, as observed for ALS. However, if the heap space is insufficient after a GC cycle, another cycle will be triggered with the same ID. As Metronome supports class unloading in the standard way, there might be pause time outliers during GC activities, inducing a negative impact on the Neo4j execution time and energy consumption.

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

Finally, the results of HOTSPOT shared similarities with GRAALVM. The ParallelGC happened to give better (6% for Dotty) or worst (10% for Neo4j) energy efficiency compared to the default GC. On the other hand, ParallelOldGC and Serial GC gave better results than the default G1 GC. More specifically, the second one consumed 30% and 6% less energy than 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 a 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 reduce the 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 a configuration to another. The default GC consumed more energy than other strategies in most of the situations. However, keeping the default JIT parameters often delivers near-optimal energy efficiency. In addition, the JVM platforms can handle differently multi-threaded applications and thus consume a different amount of time/energy. Dedicated performance tuning evaluations should therefore be conducted on such software to identify the most energy-efficient platform and settings.

6.4 Threats to Validity

A number of 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 [44, 23]. However, RAPL only gives the global energy consumption and no fine-grained measures at process or thread levels. We used baremetal hardware with a minimal OS and tuned 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 [68].

Another measurement issue is the CPU energy variation within machines [68], 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 on 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 on how long the benchmark is running for the hardware we used,

Table 6.8: J-Referral recommendations.

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

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

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

6.5 Tools and contributions

Table 6.8 illustrates an example of the final report of J-Referral. The tool was tested for 2 real Java projects: Zip4J and[10] and K-nucleotide.[11]. Zip4J runs a large file compression, while K-nucleotide extracts a DNA sequence, and updates a hashtable of k-nucleotide keys to count specific values. The short report presented in Table 6.8 shows the ratio of potential energy saving between the most and least energy consuming tested JVM (40% and 70% energy savings for Zip4J and K-nucleotide respectively). Options are available for J-Referral to obtain much more detailed reports including execution time, DRAM usage, split DRAM vs. CPU consumption, etc. The tool is available as *open-source software* (OSS) from our anonymous repository.[12]

6.6 Conclusion

The results of our investigations showed that many JVMs share energy efficiencies and can grouped into 3 classes: HOTSPOT, J9, and GRAALVM. The 3 selected JVM classes can however report a different energy efficiency for different software and/or workloads,

[10]https://github.com/srikanth-lingala/zip4j
[11]https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/knucleotide.html
[12]https://github.com/chakib-belgaid/jreferral

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

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

Chapter 7

Discussion and Conclusion

7.1 Conclusion

7.2 Summary of Contributions

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

7.4 Future Work

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

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

Bibliography

[noa] PYPL PopularitY of Programming Language index.

[2] (2008). Pearson's correlation coefficient. In *Encyclopedia of Public Health*. Springer Netherlands.

[3] Acun, B., Miller, P., and Kale, L. V. (2016). Variation among processors under turbo boost in hpc systems. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–12.

[4] Balasubramanian, N., Balasubramanian, A., and Venkataramani, A. (2009). Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, pages 280–293.

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

[6] Banerjee, A. and Roychoudhury, A. (2016). Automated re-factoring of android apps to enhance energy-efficiency. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 139–150.

[7] Baskar, P., Joseph, M. A., Narayanan, N., and Loya, R. B. (2013). Experimental investigation of oxygen enrichment on performance of twin cylinder diesel engine with variation of injection pressure. In *2013 International Conference on Energy Efficient Technologies for Sustainability*, pages 682–687. IEEE.

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

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

[10] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development

and analysis. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA. ACM Press.

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

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

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

[14] Burtscher, M., Zecena, I., and Zong, Z. (2014). Measuring GPU power with the K20 built-in sensor. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, pages 28–36.

[15] Calero, C., Mancebo Pavón, J., and García, F. (2021). Does maintainability relate to the energy consumption of software?

[16] Chamas, C. L., Cordeiro, D., and Eler, M. M. (2017). Comparing REST, SOAP, Socket and gRPC in computation offloading of mobile applications: An energy cost analysis. In *2017 IEEE 9th Latin-American Conference on Communications (LATINCOM)*, pages 1–6. IEEE.

[17] Chasapis, D., Casas, M., Moretó, M., Schulz, M., Ayguadé, E., Labarta, J., and Valero, M. (2016). Runtime-guided mitigation of manufacturing variability in power-constrained multi-socket numa nodes. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–12.

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

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

[20] Corral, L., Georgiev, A. B., Sillitti, A., and Succi, G. (2014). Method reallocation to reduce energy consumption: An implementation in android os. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1213–1218.

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

[22] de Matos, F. F. S., Rego, P. A., and Trinta, F. A. M. (2021). An empirical study about the adoption of multi-language technique in computation offloading in a mobile cloud computing scenario. In *CLOSER*, pages 207–214.

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

[24] Echtler, F. and Häußler, M. (2018). Open source, open science, and the replication crisis in HCI. Association for computing machinery, new york, NY, USA, 1–8.

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

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

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

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

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

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

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

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

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

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

[35] Hackenberg, D., Schöne, R., Ilsche, T., Molka, D., Schuchart, J., and Geyer, R. (2015). An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 896–904. IEEE.

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

[37] Heinrich, F., Carpen-Amarie, A., Degomme, A., Hunold, S., Legrand, A., Orgerie, A.-C., and Quinson, M. (2017a). Predicting the performance and the power consumption of MPI applications with SimGrid.

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

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

[40] Inadomi, Y., Patki, T., Inoue, K., Aoyagi, M., Rountree, B., Schulz, M., Lowenthal, D., Wada, Y., Fukazawa, K., Ueda, M., et al. (2015). Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE.

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

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

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

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

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

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

[47] Lafond, S. and Lilius, J. (2006). An energy consumption model for an embedded java virtual machine. In *International Conference on Architecture of Computing Systems*, pages 311–325. Springer.

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

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

[50] Lengauer, P., Bitto, V., Mössenböck, H., and Weninger, M. (2017). A comprehensive java benchmark study on memory and garbage collection behavior of DaCapo, DaCapo scala, and SPECjvm2008. In Binder, W., Cortellessa, V., Koziolek, A., Smirni, E., and Poess, M., editors, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 3–14. ACM.

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

[52] Lilja, D. J. (2005). *Measuring Computer Performance: A Practitioner's Guide*. Cambridge university press.

[53] Liu, K., Pinto, G., and Liu, Y. D. (2015). Data-oriented characterization of application-level energy optimization. In *International Conference on Fundamental Approaches to Software Engineering*, pages 316–331. Springer.

[54] Ma, H., Simon, D., Siarry, P., Yang, Z., and Fei, M. (2017). Biogeography-based optimization: A 10-year review. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(5):391–407.

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

[56] Marathe, A., Zhang, Y., Blanks, G., Kumbhare, N., Abdulla, G., and Rountree, B. (2017). An empirical survey of performance and energy efficiency variation on Intel processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, pages 1–8.

[57] Mirowski, P., Mathewson, K., Branch, B., Winters, T., Verhoeven, B., and Elfving, J. (2020). Rosetta code: Improv in any language. In *Proceedings of the 11th International Conference on Computational Creativity*, pages 115–122. Association for Computational Creativity.

[58] Mishra, S. and Srivastava, S. (2021). Web development frameworks and its performance analysis—a review. *Smart Computing*, pages 337–343.

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

[60] Mukherjee, T., Banerjee, A., Varsamopoulos, G., Gupta, S. K., and Rungta, S. (2009). Spatio-temporal thermal-aware job scheduling to minimize energy consumption in virtualized heterogeneous data centers. *Computer Networks*, 53(17):2888–2904.

[61] Murri, R. (2013). Performance of Python runtimes on a non-numeric scientific code. page 6.

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

[63] Nanz, S. and Furia, C. A. (2015). A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788. IEEE.

[64] Noureddine, A. (2022). PowerJoular and JoularJX: Multi-platform software power monitoring tools. In *18th International Conference on Intelligent Environments*.

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

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

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

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

[69] Pankiv, A. (2019). *Concurrent Benchmark System for Web-Frameworks on Python*. PhD thesis, Ukrainian Catholic University.

[70] Peng, R. D. (2011). Reproducible research in computational science. *Science (New York, N.Y.)*, 334(6060):1226–1227.

[71] Pinto, G., Liu, K., Castor, F., and Liu, Y. D. (2016). A comprehensive study on the energy efficiency of java's thread-safe collections. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 20–31. IEEE.

[72] Prokopec, A., Rosà, A., Leopoldseder, D., Duboscq, G., Tuma, P., Studener, M., Bulej, L., Zheng, Y., Villazón, A., Simon, D., Würthinger, T., and Binder, W. (2019). Renaissance: Benchmarking suite for parallel applications on the JVM. In *PLDI*, pages 31–47, Phoenix, AZ, USA. ACM.

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

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

[75] Sonnenwald, D. H., Whitton, M. C., and Maglaughlin, K. L. (2003). Evaluating a scientific collaboratory: Results of a controlled experiment. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 10(2):150–176.

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

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

[78] von Kistowski, J., Block, H., Beckett, J., Spradling, C., Lange, K.-D., and Kounev, S. (2016). Variations in cpu power consumption. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 147–158.

[79] Wang, X., Zhao, H., and Zhu, J. (1993). GRPC: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Operating Systems Review*, 27(3):75–86.

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

[81] Yet, A. W. F. (2016). Cross-language compiler benchmarking.