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 October 2021

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

Declaration

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

Mohammed Chakib Belgaid
October 2021

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

Basically we want to save the planet, and save your pockets within it. We know that humans don't care about anything else except their pockets. So let's try to push them into saving our mother earth by making them reduce their costs and increase their benifits. How to do so ? well we will work on reducing the elecricity consumption of their services. without making them to change their machines, their providers or impacting the quality of service that they provide.

Our approach is is to reduce the energy consumption of the software services by changing certains parameters, such as palteform, programming langauge, and/or the design pattern.

well to do so we have multiple approachs , 1- by using formal apporachs such as static analysers, complexity , nad programming theories – lets check maybe there is a state of the art - 2- second by a more interactive appreach where based on tests and feedback

This theis is about the second one, So basically all we do is testing programms, measuring their energy and then provide a feedback based on our observations. To do so we first need to setup a testing enviroment. something that allows us to have accurate, precise and reproductible tests. Therefore the first chapter our the thesis will be dedicated to provide green coders with a set of elements in orders to test their hypothesis.

....

1.1 Motivation

....

1.2 Research Contributions

..... This work makes the following contributions:

1. Introduces a new ...

2. Shows how ....

3. Proposes ...

1.3 Publications

1. ...

2. ...

3. ...

Chapter 2

Problem Background

Chapter 3

Literature Review

Chapter 4

Empirical Evaluation Protocol

In order to optimize the energy consumption of software we first want to setup a experemental enviroment. We have chose to use the empirical appraoch so we will reduce the energy consumption with trial and error.

4.1 Threats and Challenges

A seccusefull test has three creterions tp fullfill. First, it has to be *reproducible* in order for others to replecate. Second, the results should be *accurate*, which means each time we rerun the test we are expecting the same results. Finally, it should *represent* the real world. In other words, the conclusions brought from the experemnt should be valide outside the expermentation as well. In our case the real world is the production enviroment, Therefore our experements should reflect what is happening in the production environements. In this section we will dig deeper in each aspect, present what has been done by others and ourselfs as well, and finally we will propose some perspectives, to make the experements better .

4.1.1 Reproducibiliy

One of the most challenges which face the reaserchers is the reproducibility of their tests.Actually many results are not reproducible [1], which let to a *replication crisis*.When this crisis hit most of the empirical studies,most of the reviews now includes reproducibility as one of the minimal standard for judging scientific [16]. one of the creterions for a reproducidibility is the publication of the dataset and the algorithm run on the raw data to condlude the results. There is even some disagreement about what the terms "reproducibility" or

[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.

"replicability" by themselves mean [9] .According to [5] *replicability* extends *reproducibility* to the ability to collect a new raw dataset comparable to the original one by reexecuting the experiment under similar condition, Instead of just the ability to get the same results by ruruning the statistical analyses on the orginal data set.

In this area, 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 on homogeneous nodes, but even within a single node.

One major problem that hinders the reproducibility of the empirical tests is the interaction with the external environment, either as concurency or dependencies. Therefore testers won't have the same results unless they duplicate the same environment.

4.1.2 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". Compare with precision. In our 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 the measures. 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 enought to reproduce the the same energy measures,not only between identical machines but even within the same machine. This difference—also called *energy variation* (EV)—has become a serious threat to the accuracy of experimental evaluations.

Figure 4.1 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 4.1) at 50 % workload. we can observe a large variation of the energy consumption, not only among homogeneous machiness, but also at the scale of a single machines, reaching up to 25 % in this example.

Some reaserchers started investigating the hardware impact of the energy variation of power consumption. As an example we cite [2, 22] who reported that the main cause of

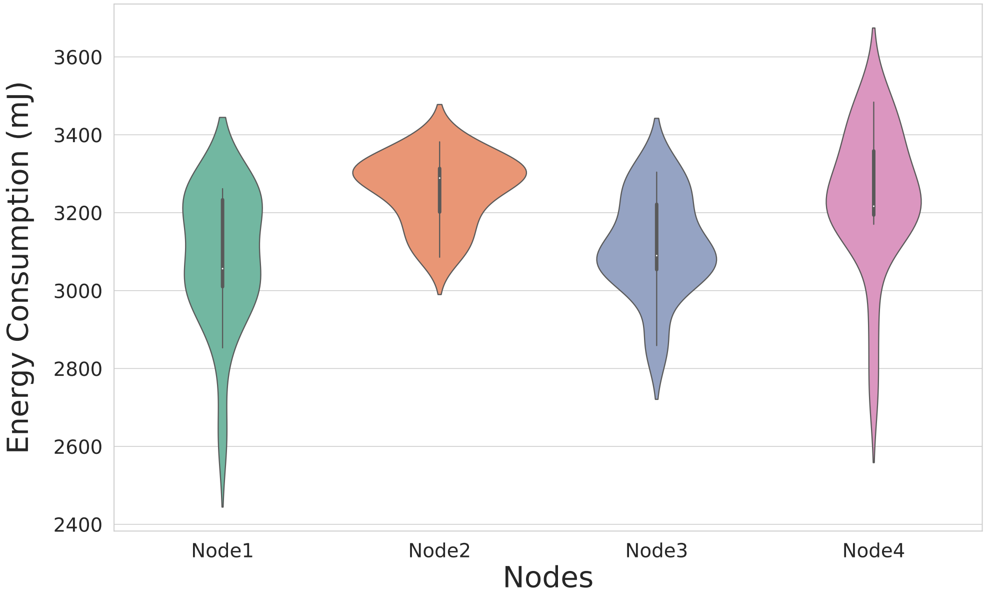


Figure 4.1: CPU energy variation for the benchmark CG

the variation of the power consumption between different machines is due to the **CMOS** manufacturing process of transistors in a chip. [**?**]. decribed this variation as a set of parameters such as CPU Frequency and the termal effect.

4.1.3 representativeness

As obvious as it seems, the reason of the doing tests is to validate ideas so we can use them in the real life. Howerver this is means that those tests have to repesent reality somehow. Basically when we want to test something we create a mock up version of the situation that we want it to wrok. But how can we assure that the test is representative ?. honestly i don't know. and we can't generalize this. but there are some works that have been done for this. First we can talk about the benchmarking and their selection, then we will talk about the tress test for some applications. and finally we will bring this representativeness in our case and how can we get closer to the energy consumption behaviour of our software between the test environement and the production one.

As Stephen M. Blackburn et al cited in their paper "evaluate collaboratory" [**?**] one of major pitfalls of the measurement contexts is the inconsistancy which is translated here with the fact that the production context is not the same as the testing one.

another difficult part for practicioners to generlize their claimes that they have got in the lab outside is the the the workloads. are they appripriate ? are they consistent and are they reproducible ?. To answer those questions the community agreed on some well known benchmarks that they represents an aspect of the the production world. we can site as an exemple the Dacapo set and Renaissance for JAVA applciations. the CLBG benchmark set for programming langauges. Although they do not cover all the cases. the community agrees on them. for the moment

In addition to those benchmarks a new category of testings has been born to simulate the worst case of the production enviromenet, Stress tests, are tests meant to evaluate the behaviour of a software under extreme situations. we can cite as an exemple Gatling for web application and stress-ng for hardwares.

Studying Hardware Factors. This variation has often been related to the manufacturing process [4], 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 [13] 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 [24] claimed that the processor thermal effect is one of the most contributing factors to the energy variation, and the correlation between the CPU temperature and the energy consumption variation is very tight.

TODO : add the corelation value

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

The ambient temperature was also discussed in many papers as a candidate factor for the energy variation of a processor. In [23], 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 [7], 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 [24], 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.* [12] used three different power measurement tools: RAPL, Power Insight[2] and BGQ EMON. All of the three tools recorded the same 10 % of energy variation, that was supposedly related to the manufacturing process.

Mitigating Energy Variations. Acknowledging the energy variation problem on processors, some papers proposed contributions to reduce and mitigate this variation. In [12], the authors introduced a variation-aware algorithm that improves application performance under a power constraint by determining module-level (individual processor and associated DRAM) power allocation, with up to 5.4× speedup. The authors of [10] proposed parallel algorithms that tolerate the variability and the non-uniformity by decoupling per process communication over the available CPU. Acun *et al.* [1] 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 slow 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 [3], 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.

Our position

Meanwhile the previoius work was to create a large umbrella tht englobes all emprical tests related to computer science, we want to narrow our study to energy testing only. In other words we will study pitfalls that hinders the energy claimes of software .

We are fully aware of the impact that hardware has on energy variation. However, we believe that there is still a room to reduce this energy variation for practicioners using only paramenters that they are in cotrol. 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 tests. We will first start with evaluating the paramenters mentioned by [**?**].

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

4.1.4 Virtualisation

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 [11] 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 .

4.1.5 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.

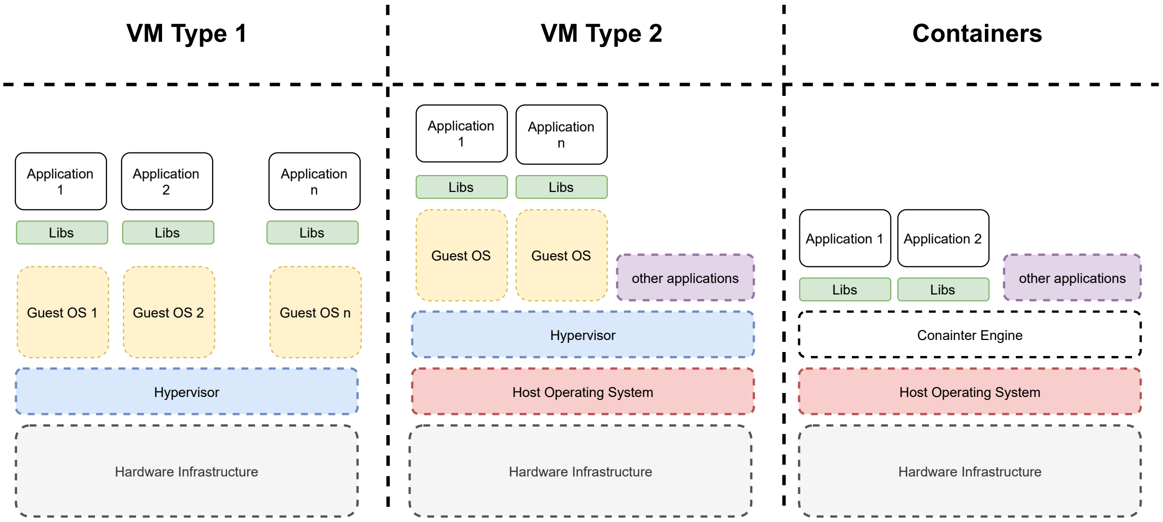


Figure 4.2: Different Methods of Virtualzation

Figure 4.2 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 reasearchers 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* [**?**]

4.1.6 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.

4.1.7 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 [Eddie Antonio Santos et al.], 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 4.3 Docker braught arround 1000joules overhead. In the other hand as we can see in figure 4.4. 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.
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

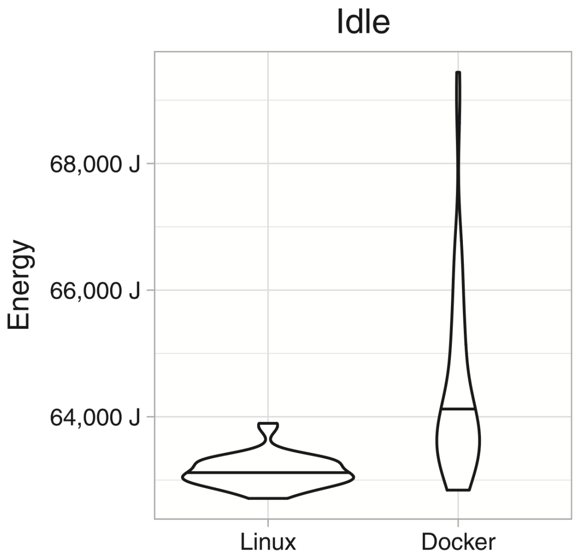


Figure 4.3: energy consumption of Idle system with and without docker [Eddie Antonio Santos et al.]

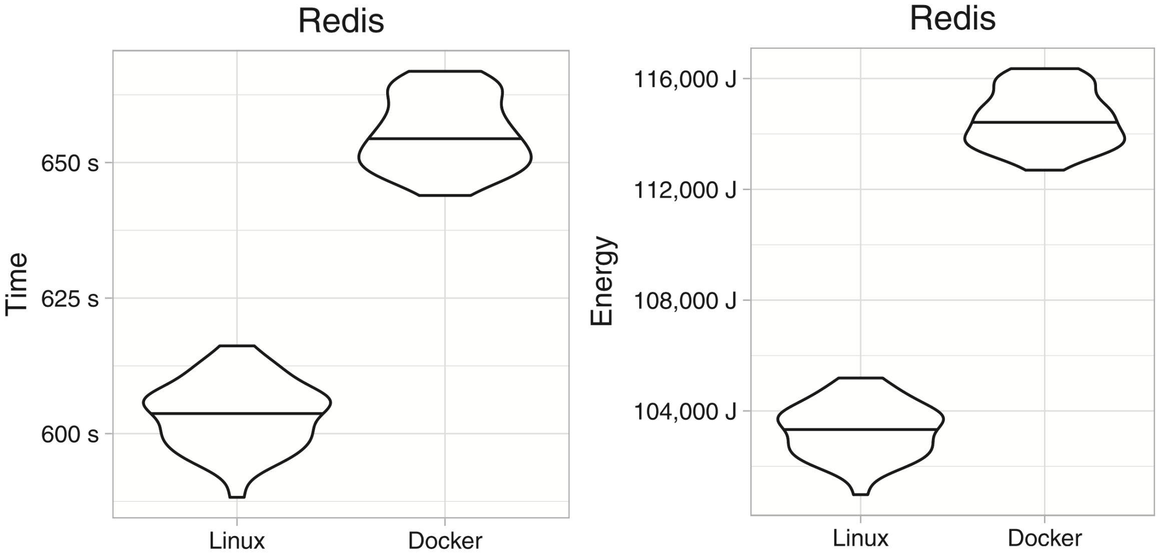


Figure 4.4: execution time and energy consumption of Redis with and without docker [Eddie Antonio Santos et al.]

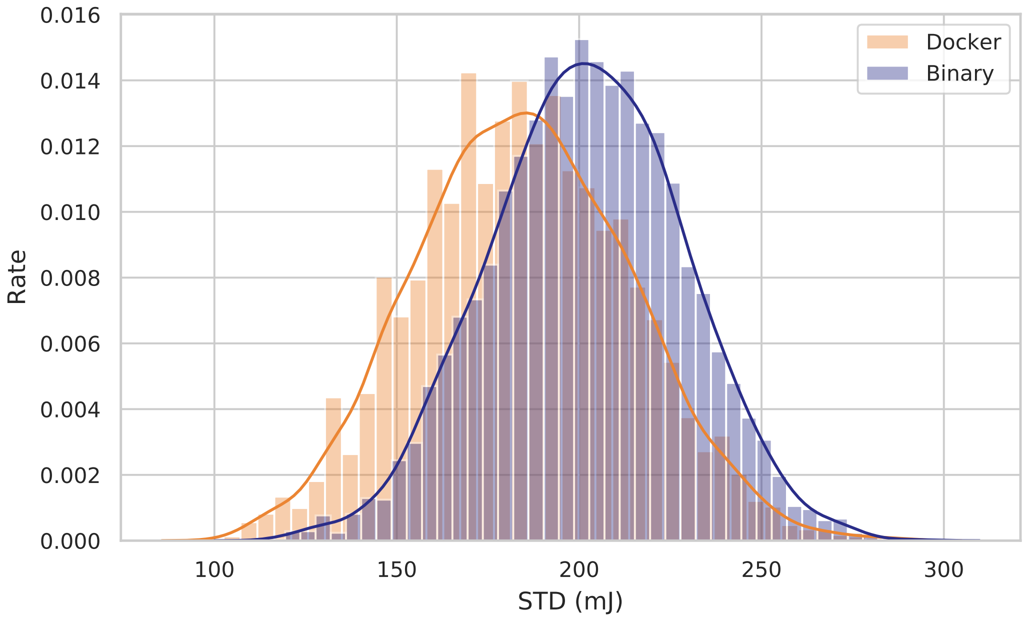


Figure 4.5: Comparing the variation of binary and Docker versions of aggregated LU, CG and EP benchmarks

4.1.8 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 4.5 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 [**?**], 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.

4.1.9 Goal

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?

4.1.10 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) [**?** **?**],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 4.1, presents a description of the 4 clusters

Table 4.1: 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 4.6 illustrates a detailed topology of a node belonging to the cluster Dahu.

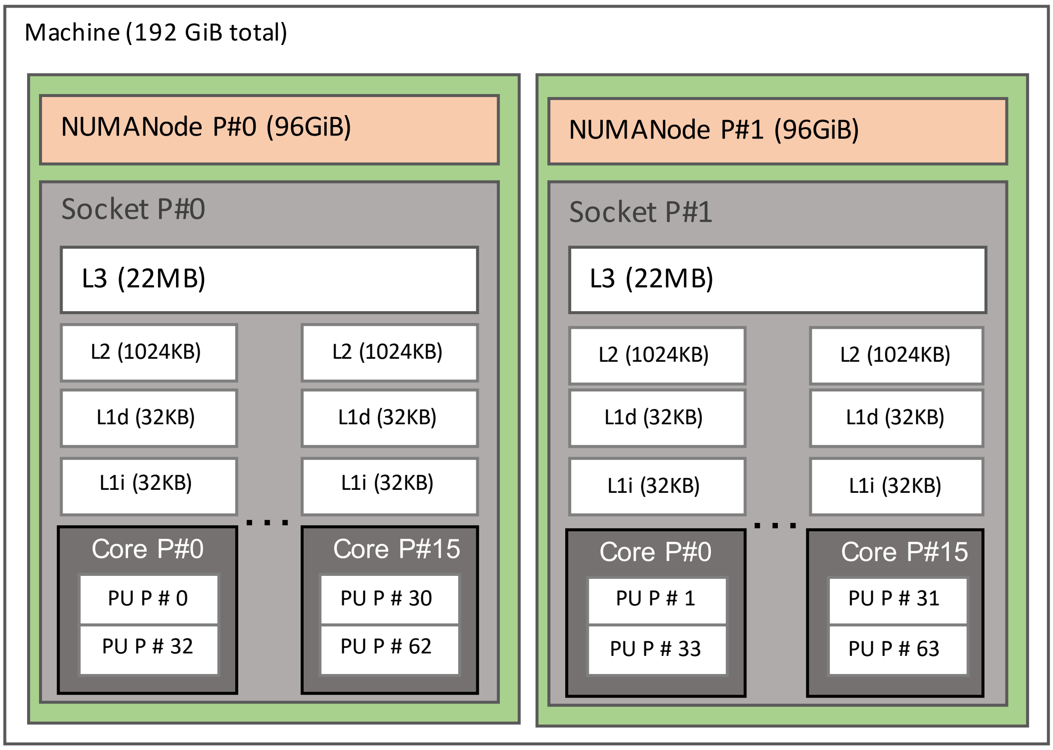


Figure 4.6: Topology of the nodes of the cluster Dahu

4.1.11 Workload

Our choice for the benchmarks was based on two creterions.

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 [**?**]. 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`,[3] `pbzip2 v1.1.9`,[4] `linpack`[5] and `sha256 v8.26`[6].

[3] https://kernel.ubuntu.com/~cking/stress-ng
[4] https://launchpad.net/pbzip2/
[5] http://www.netlib.org/linpack
[6] https://linux.die.net/man/1/sha256sum

4.1.12 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 [**?**], which is a power monitoring toolkit that is based on *Intel Running Average Power Limit* (RAPL) [**?**]. The advantage of PowerAPI is that it reports the Energy consumption of CPU and DRAM at a socket level.

Our testbeds are run with a minimal version of Debian 9 (4.9.0 kernel version)[7] 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 [**?**] to randomly build multiple subsets of data from the original dataset, and we draw the STD density of those sets, as illustrated in Figure 4.5. 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.[8] 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.

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

4.1.13 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.

4.1.14 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 4.7 reports on 300 aggregated executions of the benchmarks LU, CG and EP, on 4 machines of the cluster Dahu (cf. Table 4.1) 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 4.7. As one can observe, picking one of these strategies does not have a strong impact on the energy variation for most workloads. In fact, all the strategies seem to exhibit the same variation with all the workloads we considered—*i.e.*, the STD is tightly close between the three modes. The only exception is the reboot mode at 100 % load, where

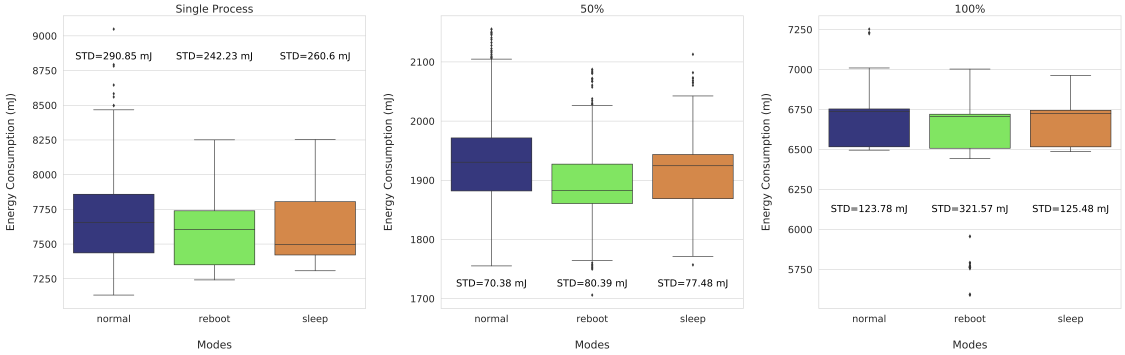


Figure 4.7: Energy variation with the normal, sleep and reboot modes

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

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

4.1.15 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 4.9 depicts the results of the experiments we executed on three nodes of the cluster Dahu. On each node, we ran the same set of benchmarks with two modes: C-states on, which is the default mode, and C-states off. Each iteration includes 100 executions of the same benchmark at a given workload, with three workload levels. We note that our results have been confirmed with the benchmarks LU, CG and EP.

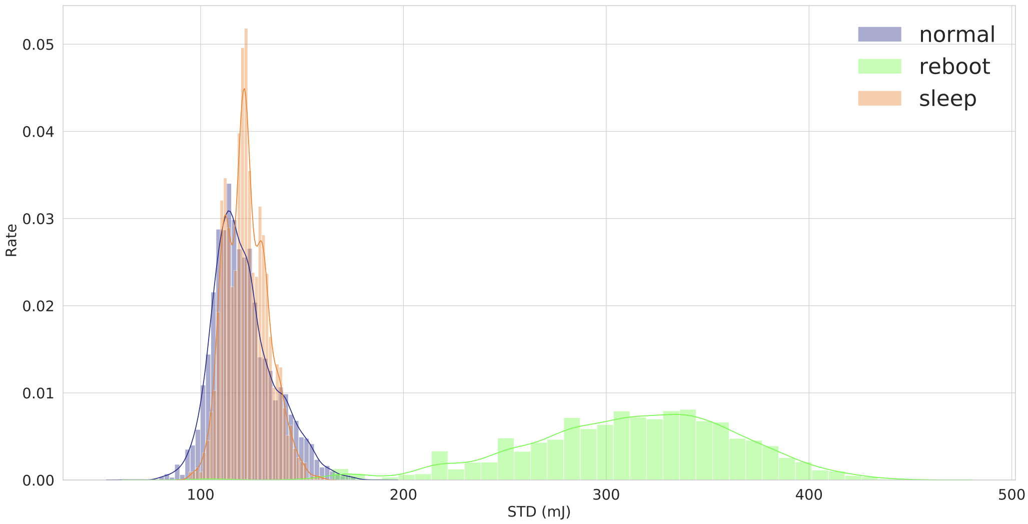


Figure 4.8: STD analysis of the normal, sleep and reboot modes

We can clearly see the effect that has the C-states off mode when running a single-process application/benchmark. The energy consumption varies 5 times less than the default mode. In this case, only one CPU core is used among 2 × 16 physical cores. The other cores are switched to a low consumption state when C-states are on, the switching operation causes an important energy consumption difference between the cores, and could be affected by other activities, such as the kernel activity, causing a notable energy consumption variation. On the other hand, switching off the C-states would keep all the cores—even the unused ones—at a 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 4.9, but reduces the variation by 5 times compared to the C-states on mode. This leads to mainly two questions: Can a process pinning method reduce/increase the energy variation? And, how does the energy consumption variation evolve at different PU usage level?

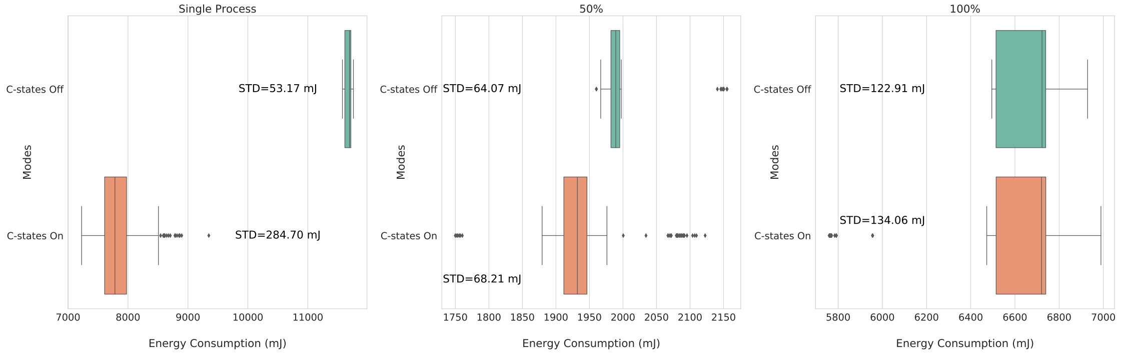


Figure 4.9: Energy variation when disabling the C-states

Cores Pinning

To answer the first question, we repeated the previous test at 50 % workload. In this experiment, we considered three cores usage strategies, the first one (S1) would pin the processes on all the PU of one of the two sockets (including hyper-threads), so it will be used at 100 %, and leave the other CPU idle. The second strategy (S2) splits the workload on the two sockets so each CPU will handle 50 % of the load. In this strategy, we only use the core PU and not the hyper-threads PU, so every process would not share his core usage (all the cores are being used). The third strategy (S3) consists also on splitting the workload between the two sockets, but considering the usage of the hyper-threads on each core—*i.e.*, half of the cores are being used over the two CPU. Figure 4.10 reports on the energy consumption of the three strategies when running the benchmark `CG` on the cluster `Dahu`. We can notice the big difference between these three execution modes that we obtained only by changing the PU pinning method (that we acknowledged with more than 100 additional runs over more than 30 machines and with the benchmarks `LU` and `EP`). For example, S2 is the least power 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.

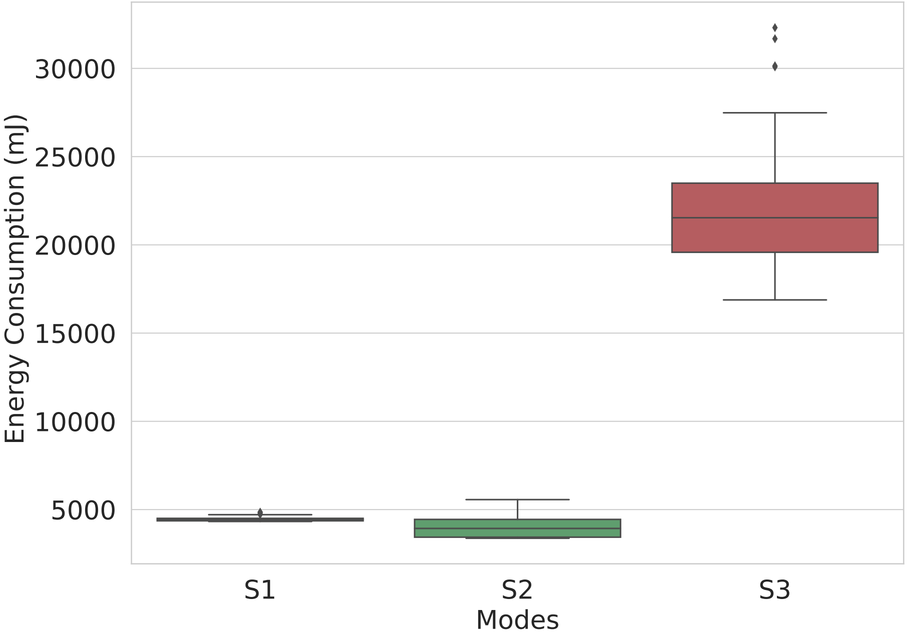


Figure 4.10: Energy variation considering the three cores pinning strategies at 50 % workload

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

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 4.6).

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 4.2: 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

Table 4.2 reports on additional aggregated results for the STD comparison on four other nodes of the cluster Dahu at 50 %, with the benchmarks LU, CG and EP. In fact, the CPU usage strategy S1 is by far the experimentation mode that gave the least variation. The STD is almost 5 times better than the strategy S2, but is up to 10 % more energy consuming ($Mean_{S1} = 4469mJ$, $Mean_{S2} = 4016mJ$). On the other hand, the strategy S3 is the worst, where the energy consumption can be up to 5 times higher than the strategy S2 ($Mean_{S2} = 4016mJ$, $Mean_{S3} = 21645mJ$) and the variation is much worst (30 times compared to the first strategy). These results allow us to have a better understanding of the different processes-to-PU pinning strategies, where isolating the workload on a single CPU is the best strategy. Using the hyper-threads PU on multiple sockets seems to be a bad recommendation, while keeping the hyper-threading enabled on the machine is not problematic, as long as the processes are correctly pinned on the PU. Our experiments show that running one hyper-thread per core is not always the best to do, at the opposite of the claims of [**?**].

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 4.11 compares the aggregated energy variation when the C-states are on and off using 2, 4 and 8 processes for the benchmarks LU, CG and EP. This figure confirms that 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.

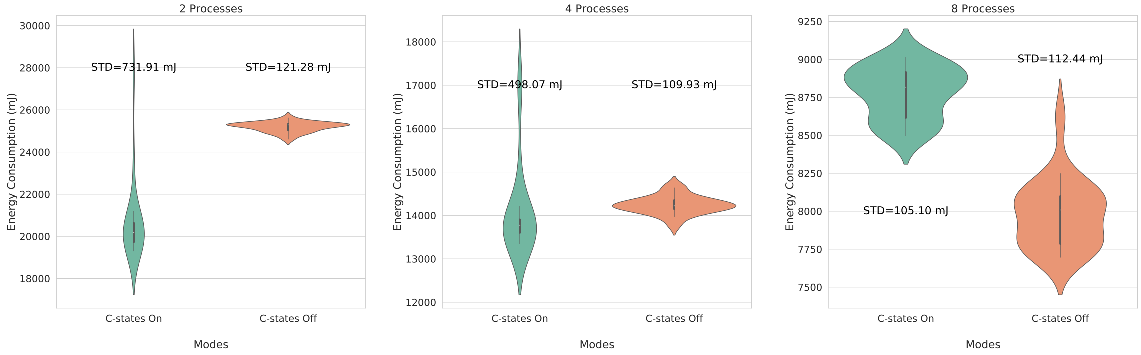


Figure 4.11: C-states effect on the energy variation, regarding the application processes count

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

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

Turbo Boost

The Turbo Boost—also known as *Dynamic Overclocking*—is a feature that has been incorporated in Intel CPU since the Sandy Bridge micro-architecture, and is now widely available on all of the Core i5, Core i7, Core i9 and Xeon series. It automatically raises some of the CPU cores operating frequency for short periods of time, and thus boost 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.* [1] 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

Table 4.3: 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

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

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

4.1.16 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

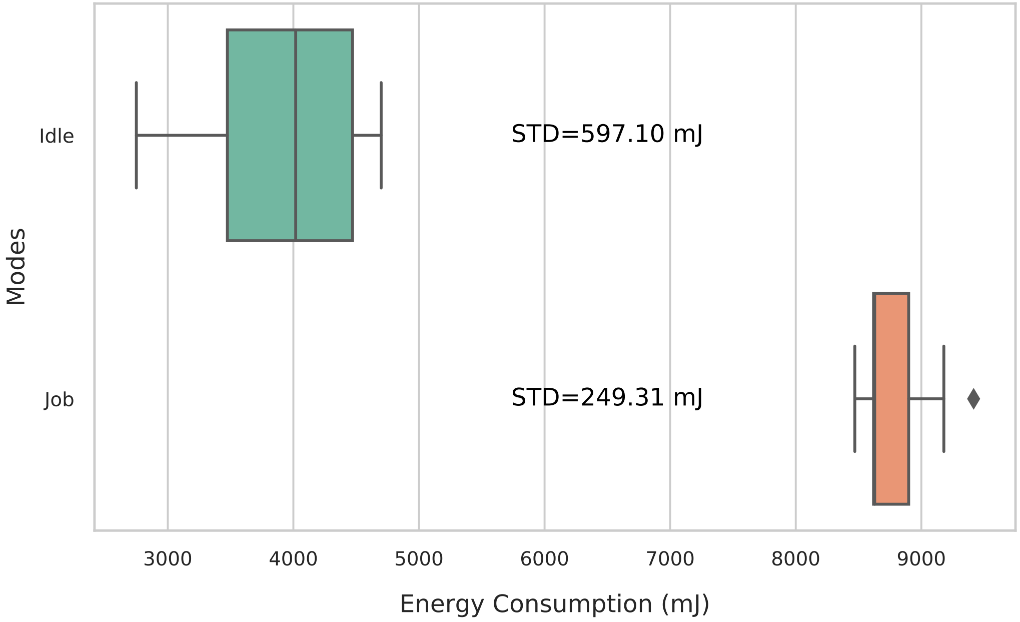


Figure 4.12: OS consumption between idle and when running a single process job

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 4.12, depict that the idle energy variation is up to 140 % worst than when running a job, even if it consumes 120 % less energy ($Mean_{Job} = 8,746mJ$, $Mean_{Idle} = 3,927mJ$). In fact, for the three nodes, randomly picked from the cluster Dahu, the idle variation is way more important than when a test was running, even if it is a single process test on a 32-cores node. This result shows that OS idle consumption varies widely, due to the lack of activity and the different CPU frequencies states, but it does not mean that this variation is the main responsible for the overall energy variation. The OS behaves differently when a job is running, even if the amount of available cores is more than enough for the OS to keep his idle behavior when running a single process.

Inspecting the OS idle energy variation is not sufficient to relate the energy variation to the active job. In fact, the OS can behave differently regarding the resource usage when running a task. To evaluate the OS and the job energy consumption separately, we used the POWERAPI toolkit. This fine-grained power meter allows the distribution of the RAPL

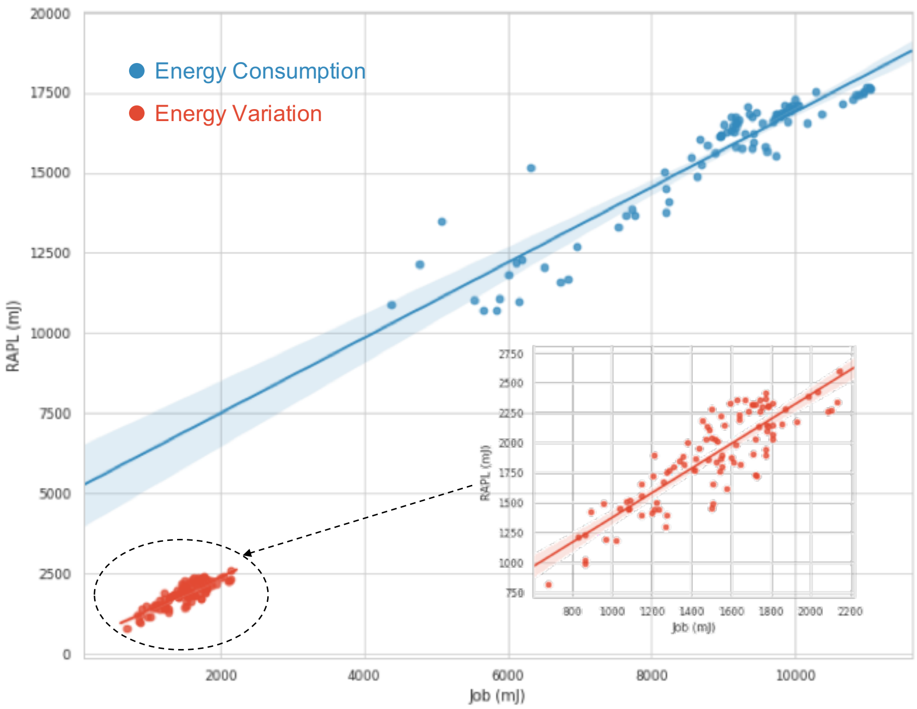


Figure 4.13: The correlation between the RAPL and the job consumption and variation

global energy across all the Cgroups of the OS using a power model. Thus, it is possible to isolate the job energy consumption instead of the global energy consumption delivered by RAPL. To do so, we ran tests with a single process workload on the cluster Dahu, and used the POWERAPI toolkit to measure the energy consumption. Then, we compared the job energy consumption to the global RAPL data. We calculated the Pearson correlation [**?**] of the energy consumption and variation between global RAPL and POWERAPI, as illustrated in Figure 4.13. The job energy consumption and variation are strongly correlated with the global energy consumption and variation with the coefficients 93.6 % and 85.3 %, respectively. However, this does not completely exclude the OS activity, especially if the jobs have tight interaction with the OS through the signals and system calls. This brings a new question on whether applying extra-tuning on a minimal OS would reduce the variation? As well as what 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.

Table 4.4: 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 %

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 4.4 reports on the aggregated results for running single process measurements with the benchmarks CG, LU and EP, on three servers of the cluster Dahu, before and after tuning the OS. Every cell contains the *STD* value before the tuning, plus/minus a ratio of the energy variation after the tuning. We notice that the energy variation varies less than 10 % after the extra-tuning. We argue that this variation is not substantial, as it is not stable from a node to another. Moreover, 10 % of variation is not a representative difference, due to many factors that can affect it as the CPU temperature or the measurement errors.

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 [**?**] 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 4.5 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.

Table 4.5: STD (mJ) comparison with/without the security patch

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

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

4.1.17 RQ 4: Processor Generation

Intel microprocessors have noticeably evolved during these last 20 years. Most of the new CPU come with new enhancements to the chip density, the maximum Frequency or some optimization features like the C-states or the Turbo Boost. This active evolution caused that different generations of CPU can handle a task differently. The aim of this expriment is not to justify the evolution of the variation across CPU versions/generations, but to observe if the user can choose the best node to execute her experiments. Previous papers have discussed the evolution of the energy consumption variation across CPU generations and concluded that the variation is getting higher with the latest CPU generations [Wang et al.?], 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 4.6 indicates the characteristics of each of the tested CPU.

Table 4.6 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

Table 4.6: 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

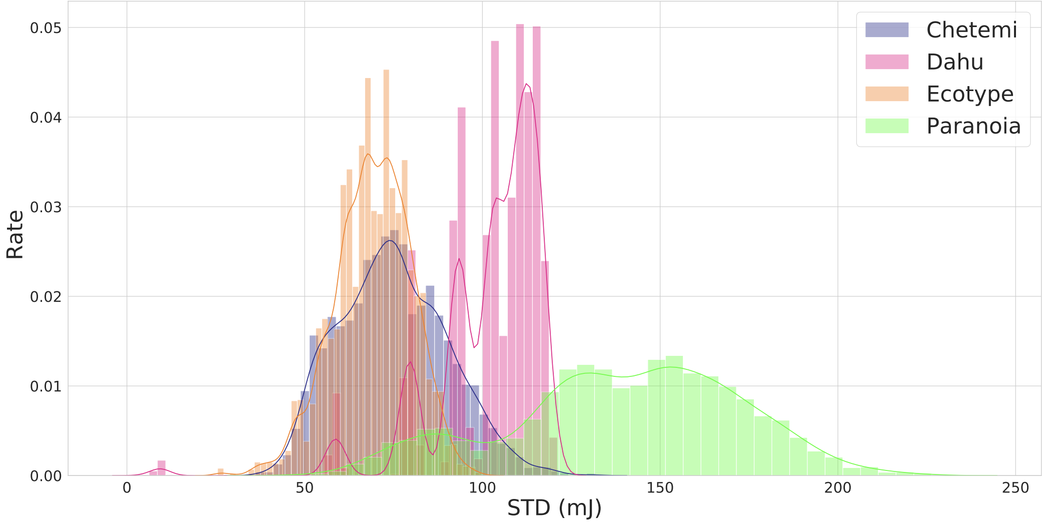


Figure 4.14: Energy consumption STD density of the 4 clusters

variation than Chetemi and the Ecotype Broadwell CPU. We argue that the hypothesis "*the energy consumption on newer CPU varies more*" could be true or not depending on the compared generations, but most importantly, the chips energy behaviors. On the other hand, our experiments showed the lowest energy variation when using the Ecotype CPU, these CPU are not the oldest nor the latest, but are tagged with "L" for their low power/TDP. This result rises another hypothesis when considering CPU choice, which implies selecting the CPU with a low TDP. This hypothesis has been confirmed on all the Ecotype cluster nodes, especially at low and medium workloads.

Figure 4.14 is an illustration of the aggregated STD density of more than 5,000-random values sets taken from all the conducted experiments. This shows that the cluster Paranoia reports the worst variation in most cases, and that Ecotype is the best cluster to consider to get the least variations, as it has a higher density for small variation values.

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

4.2 Experimental Guidelines

To summarize our experiments, we provide some experimental guidelines in Table 4.7, 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 4.7: 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×

Table 4.7 gives a proper understanding of known factors, like the C-states and its variation reduction at low workloads. However, it also lists some new factors that we identified along the analysis we conducted in Section **??**, such as the results related to the OS or the reboot mode. Some of the guidelines are more useful/efficient for specific workloads, as showed in

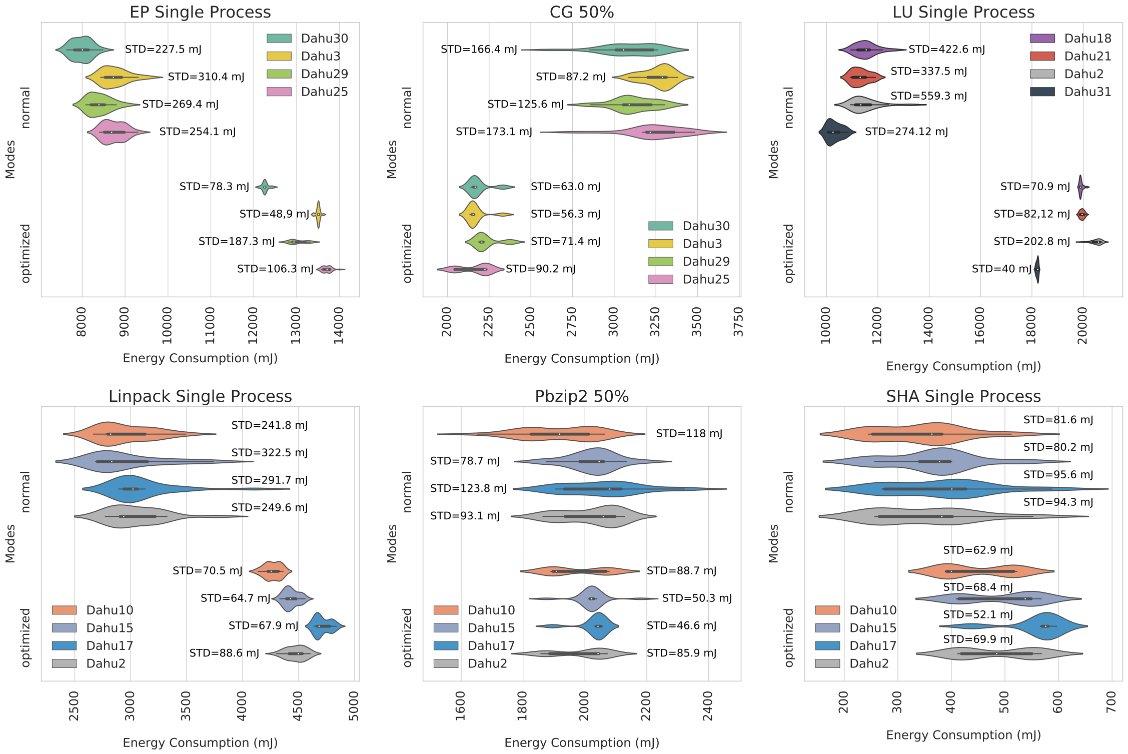


Figure 4.15: Energy variation comparison with/without applying our guidelines

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 4.15 and 4.16 highlight the improvement brought by the adoption of our guidelines. They demonstrate the intra-node STD reduction at low and medium workloads for all the

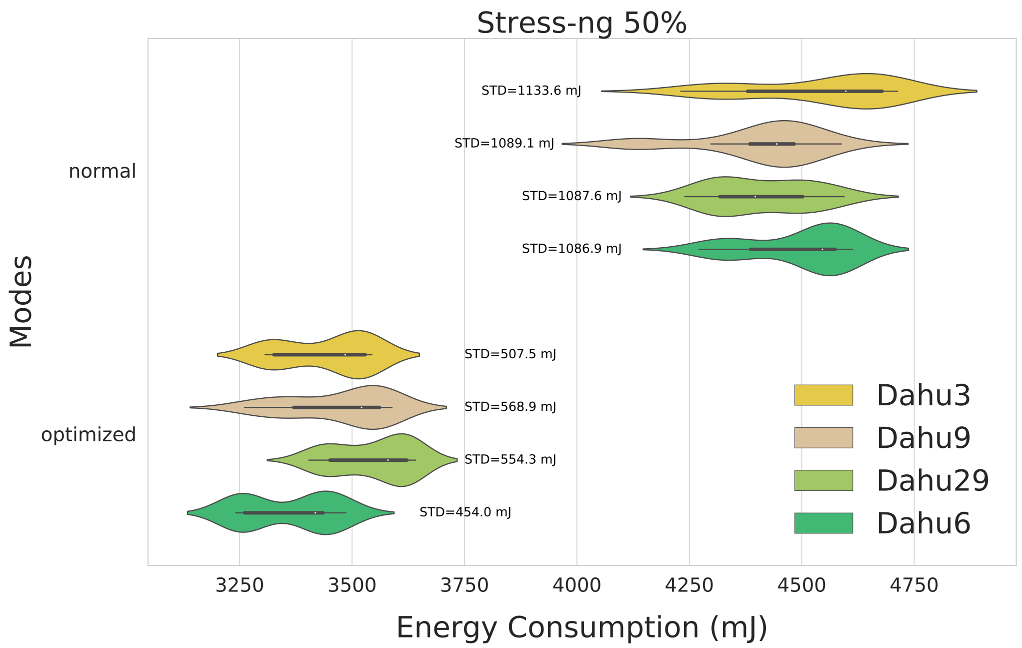


Figure 4.16: Energy variation comparison with/without applying our guidelines for STRESS-NG

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 4.15 and 4.16 also highlight that applying the guidelines does not reduce the inter-nodes variation in all the cases. This variation can be up to 30 % in modern CPU [Wang et al.]. However, taming the intra-node variation is a good strategy to identify more relevant mediums and medians, and then perform accurate comparisons between the nodes variation. Even though, using the same node is always better, to avoid the extra inter-nodes variation and thus improve the stability of measurements.

4.3 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.

4.4 Conclusion

In this paper, 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

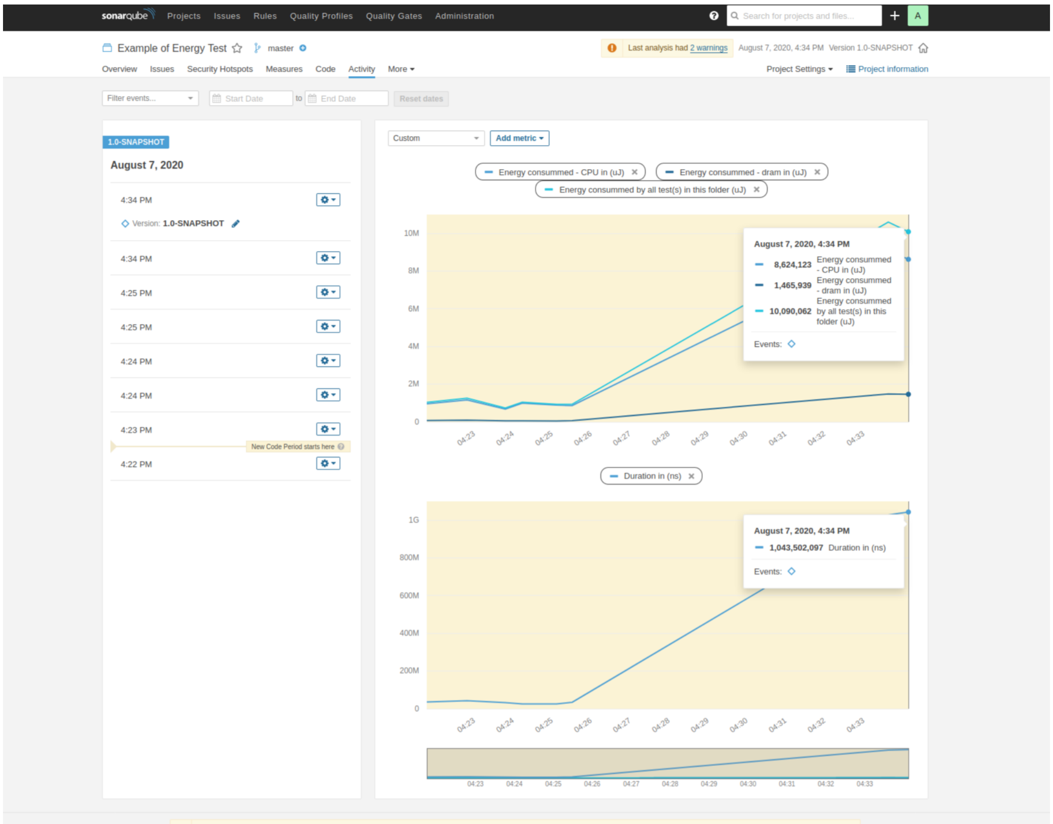


Figure 4.17: Example of the Junit Sonar Plugin

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.

4.5 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 5

The Energy footprints of programming languages

In this chapter we will dicuss the impact of the choice of the programming language on the energy consumption of the 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 reallife use case. The sections belows will provide two study cases.

5.1 Remote Procedure Call

5.1.1 Definition

5.1.2 Motivation

With the emerging technology to the cloud many protocols wanted to take the lead. Nowdays most of the architectures are based on multi-servicces and micro services. And to have higher versatility of developpers mulptile companies choose to be open to different programming langauges. basically it would be more efficient if we take advantage of each programming language to satisfy a specific need. However the callenge nowdays is to make the bridge between those plateforms. We have many initiatives such as openapi that try to create a taxonomy for rest apis. other approachs is to impelement all the different interfaces of the protocol by them selfs such as the RPC

5.1.3 State of the art

5.1.4 Research Questions

In this section we will first explore the ease of implemenation of this protocol then we will try to answer the following Research questions

RQ 1: How do RPC implementations react to the size of the request ?

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

5.1.5 Experimental protocol

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 GIB 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.

Energy measurements

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

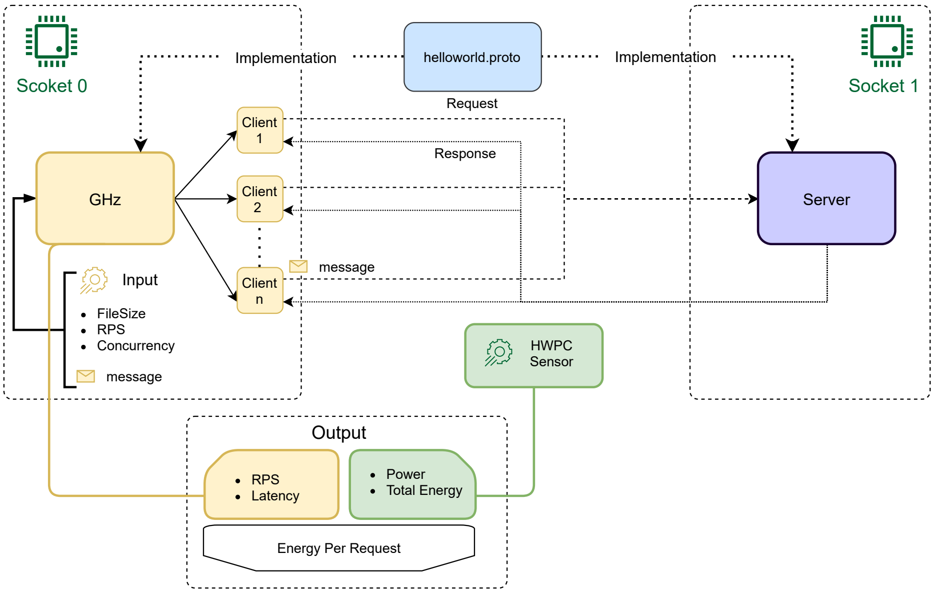


Figure 5.1: Experemenal software architecture

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 one on a different socket, in order to reduce the effect that the client might have on the server ones 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 take the whole socket for their experiment. In addition all the extra services, such as OS, hwpc, etc. are run on Socket 0. Therefore, the only process being executed in Socket 1 is the server that we benchmark.

Client : 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 get the average power for each request form both the server and the client sides. The new version is available in the repo. The client will take the protocol description to generate an implementation for the message helloworld, and then fork 50 instances that will send the same request to the server simultaneously.

Server : 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

5.1.6 Results and finding

[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 for transferring large objects. To do so, we send to the server 80,000 requests with a size scaling from 10 bytes up to 10 Megabytes, which gives 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 sec 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 :

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 :

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: where 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 (av 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:

- Green languages: CPP, GO, RUST, ELIXIR, and PHP
- Middle class: Most of the interpreted languages and vm-based ones
- Energivore 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 ressources, and sometimes less, because it takes more time to handle the less requests. This class of languages tend to be the most energivore 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.

Special mention to 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, Kotlin ;
- The size of request exceeded the maximum size by implementation : this is where the implementation could not handle requests with large size, such as .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 main modes.

- Low number of clients : where the total number of simultaneous clients is less than 100
- Moderate to high number of clients : the number of clients exceeds 100

Lite mode The implementations can be grouped into two main categories :

1. Green Frameworks: most of the framework's power consumption is around 33 Watts.
2. Energivore frameworks : where the average power consumption is higher than 37 Watts.

In each programming category we observe both behaviours green and energivore. 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 since 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 see that it is correlated with the category of the programming language rather than the implementation itself.
We can clearly highlight that after VM based languages have a significant increase in the average power consumption after they receive more than 100 simultaneous clients. This increase reaches almost double. Except PHP, all the interpreted languages preserved their energetic behaviour. Same for the compiled ones. Our Hypothesis points to the JIT, since it will compile the code and make it run faster so it will stress the CPU more.
A Remarquable behaviour has been noticed for the grallVM, is the decrease of the energy

consumption when we increase 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 couldn't 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 have three observable variables

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

Satisfaction ratio Most of the frameworks tried to satisfy all the requests, by either reducing the number of requests per second or by increasing the time of the treatment. However, there are some frameworks that have chosen a different approach such as dart or scalla where the choice was to keep a certain limit of latency even if not all the requests are answered.
Furthermore, we tend to see 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 one, and after him we see JAVA and Go in second place, on the other end Python and ruby are the least performant.

Tail LAtency However, the increase in the number of requests per second, doesn't necessarily mean a lower Latency. As we see in the 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 hungry, so we propose to investigate this trade off between the energy and the performance. Todo so we will give 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 keep a constant cost, Java, .net and go are the greenest. and Python, ruby are the most costly with 10x higher. Therefore we conclude that the number of clients won't impact the energy that much. Next study will be the payload and how the size of the requests will affect the energy consumption of the framework

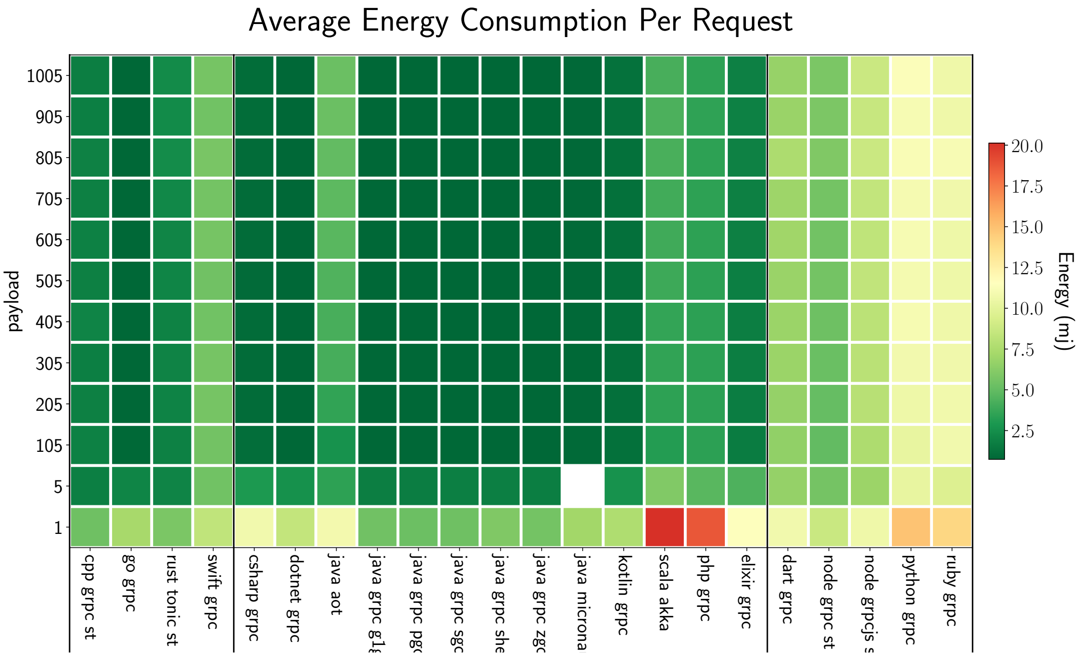


Figure 5.2: Experemenal software architecture

5.1.7 Threads to validity

5.1.8 Conclusion

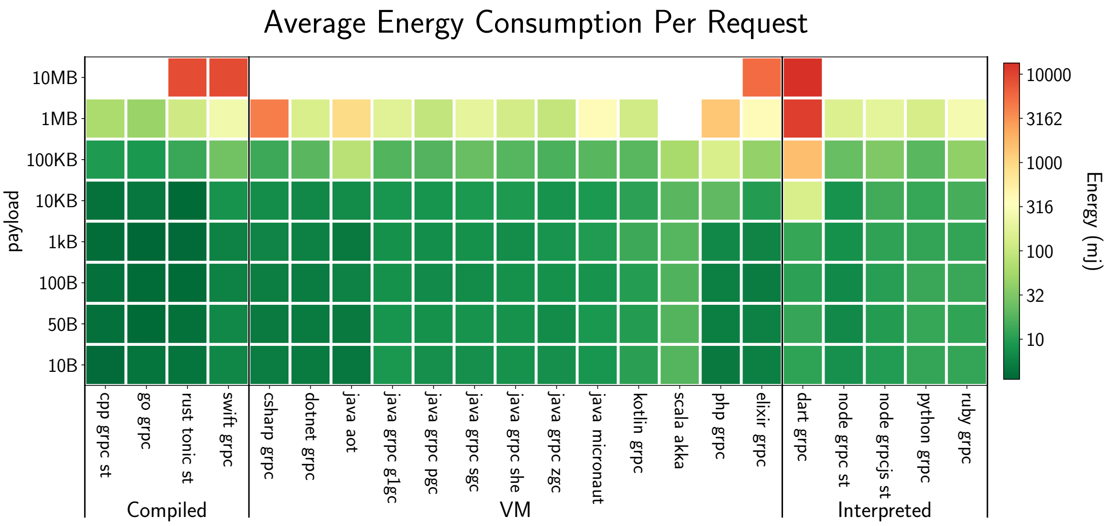


Figure 5.3: Experemenal software architecture

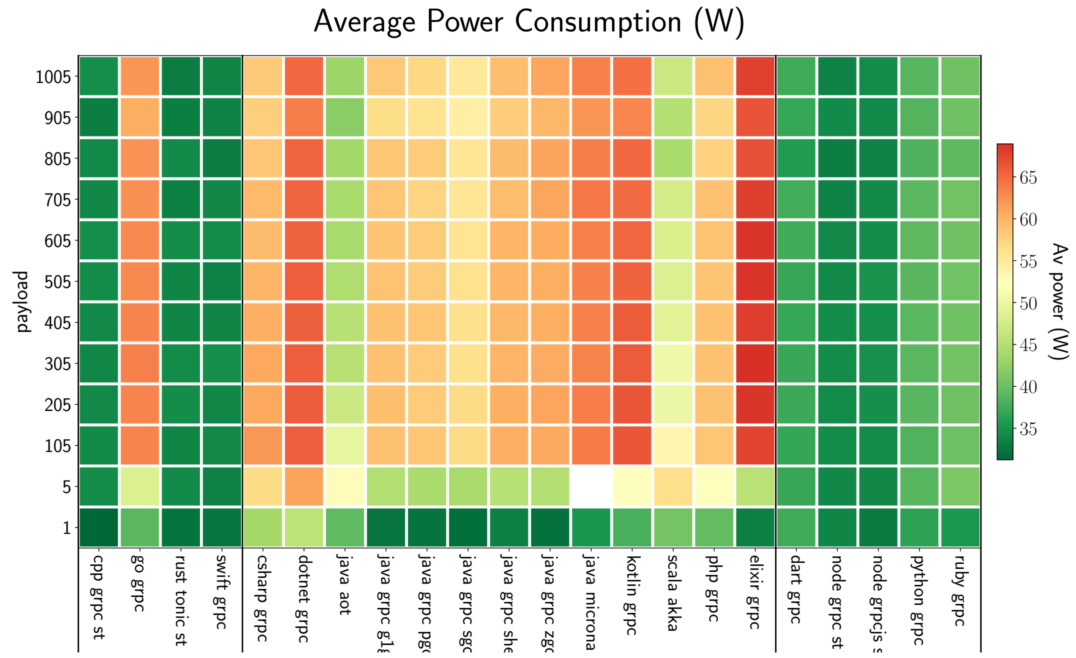


Figure 5.4: Experemenal software architecture

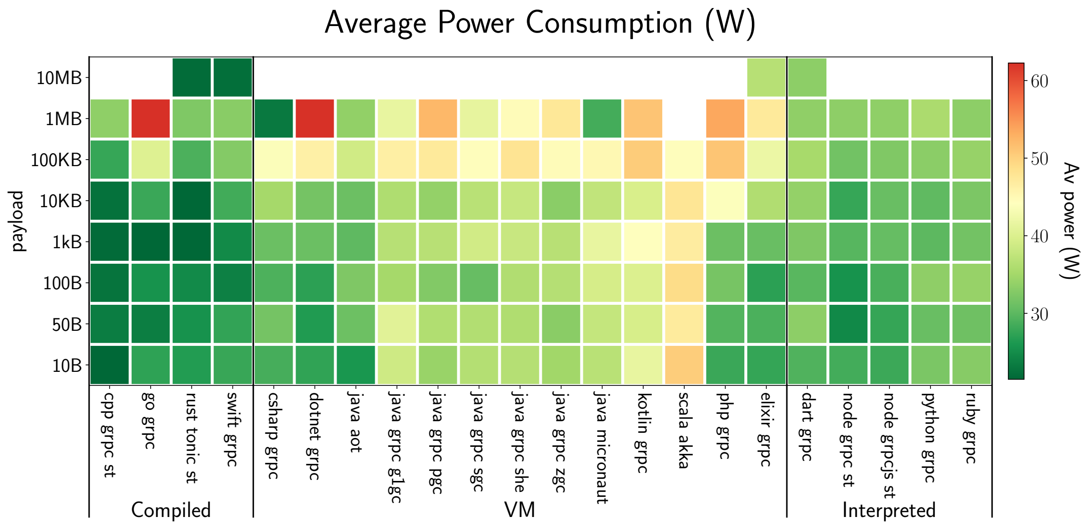


Figure 5.5: Experemenal software architecture

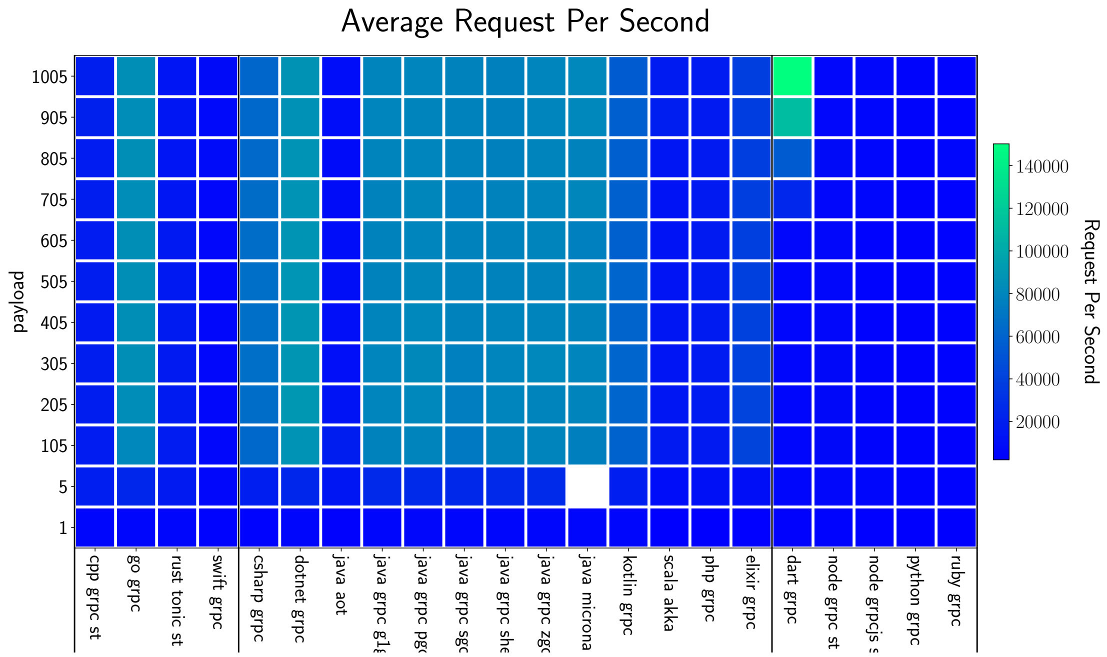


Figure 5.6: Experemenal software architecture

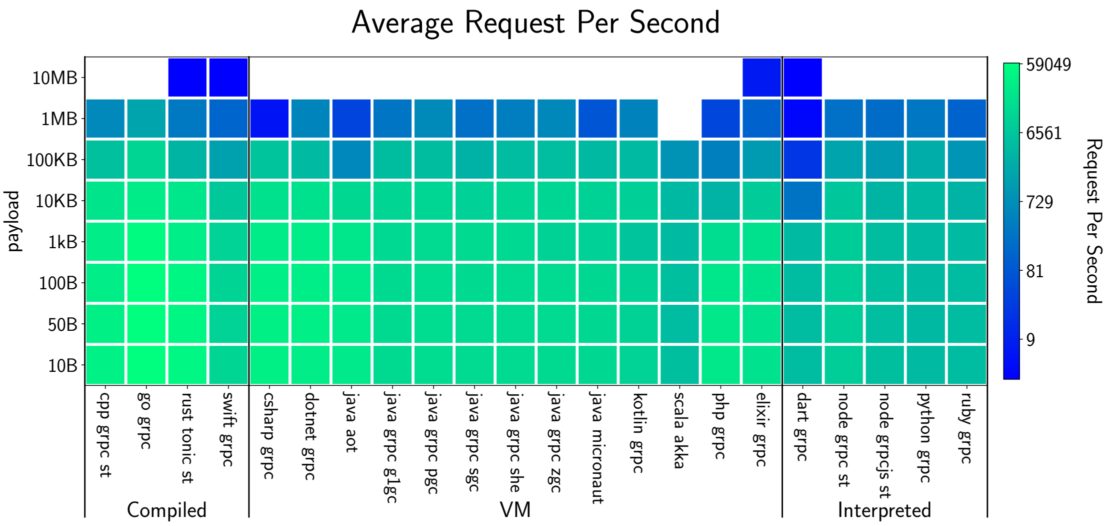


Figure 5.7: Experemenal software architecture

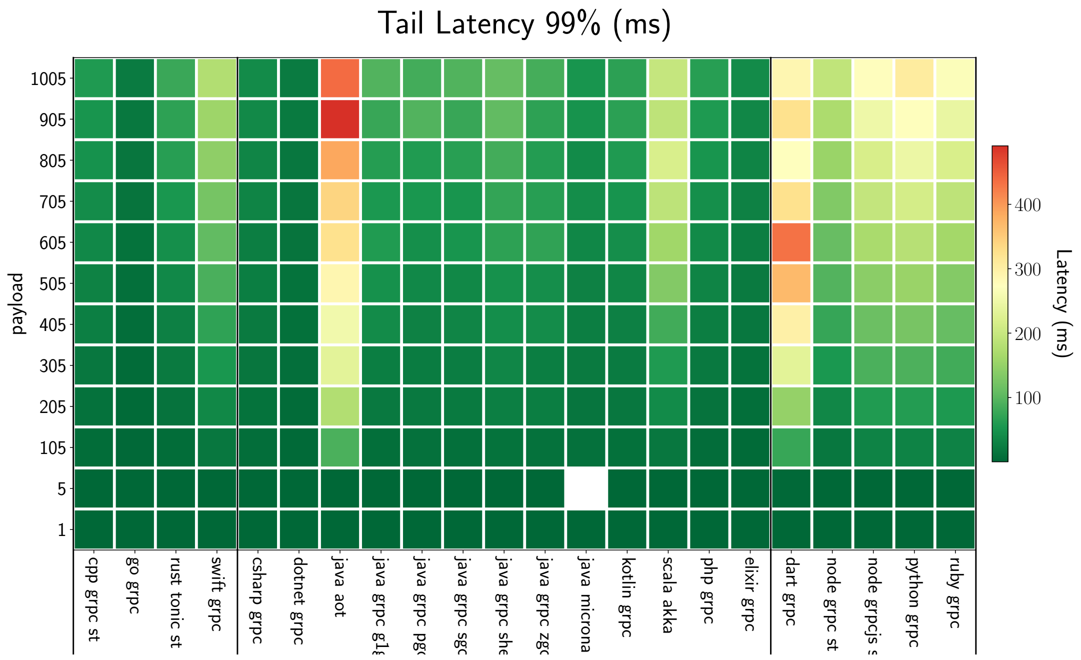


Figure 5.8: Experemenal software architecture

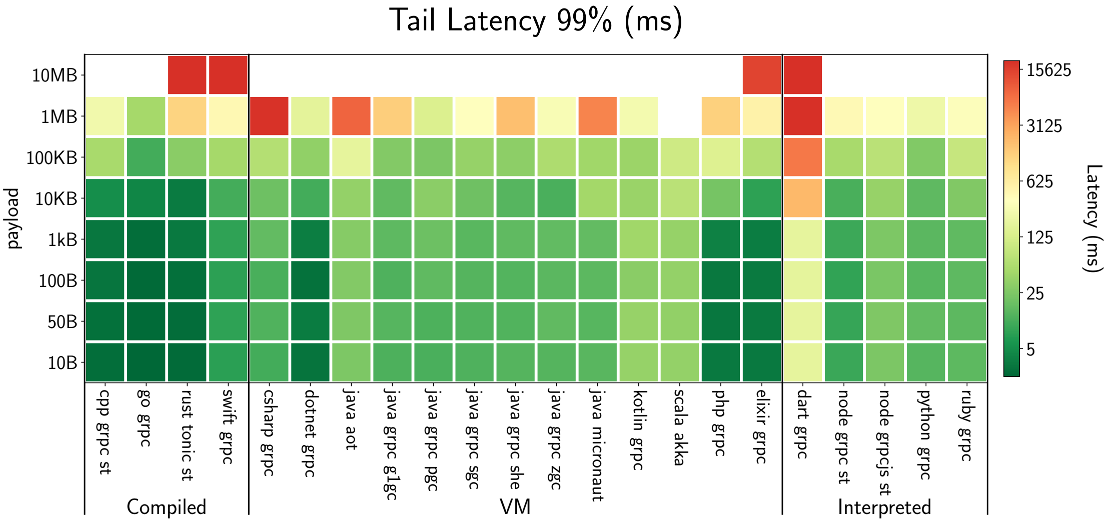


Figure 5.9: Experemenal software architecture

Chapter 6

Execution enviroment

After studying the impact of the programming languages choice on the energy consumption of softwares. we wanted to dig in more deeper. Therefore we have chosen two of the most popular programming lanaguages **Python** and **Java** In this chapter we will reduce the energy consumption of those following lanaguages by doing some tuning

Abstract

Background. The *Java Virtual Machine* (JVM) platforms have known multiple evolutions along the last decades to enhance both the performance they exhibit and the features they offer. With regards to energy consumption, few studies have investigated the energy consumption of code and data structures. Yet, we keep missing an evaluation of the energy efficiency of existing JVM platforms and an identification of the configurations that minimize the energy consumption of software hosted on the JVM.

Aims. The purpose of this paper is to investigate the variations in energy consumption between different JVM distributions and parameters to help developers configuring the least consuming environment for their Java application.

Method. We thus assess the energy consumption of some of the most popular and supported JVM platforms using 12 Java benchmarks that explore different performance objectives. Moreover, we investigate the impact of the different JVM parameters and configurations on the energy consumption of software.

Results. Our results show that some JVM platforms can exhibit up to 100% more energy consumption. JVM configurations can also play a substantial role to reduce the energy consumption during the software execution. Interestingly, the default configuration of the garbage collector was energy efficient in only 50% of our experiments.

Conclusion. Finally, we provide an OSS tool, named J-Referral that recommends an energy-efficient JVM distribution and configuration for any Java application.

6.1 Introduction

Software services are widely deployed to support our daily activities, being mobile or hosted in the cloud. Yet, beyond this undeniable success, the environmental impact of ICT is raising concerns and calls for solutions to reduce the energy footprint of software services [21].

Software developers often report that such solutions should come from more energy-efficient hardware components or optimized algorithms [24**?**] but, given the complexity of modern software environments, the composition of software layers makes this sustainability objective particularly challenging.

Given this context, this paper more specifically investigates the impact of one of these layers, the runtime environment and its settings, on the energy consumption of an hosted software service. More precisely, we aim at revealing the importance of carefully selecting and configuring the *Java Virtual Machine* (JVM) to reduce the energy consumption of any software service built from any language compatible with the Java ecosystem (Java, Kotlin, Scala, Groovy, Clojure, Jython, etc.).

The empirical study we conduct in this paper reports on the energy footprint of several versions of popular JVM distributions that are freely available for download. Beyond the choice of an appropriate runtime and its most energy-efficient version, we also consider the impact of exposed JVM settings to maximize the energy savings for a given software service. The observations of this study aim to quantify the role played by internal JVM mechanisms, like the *Just in Time* (JIT) compiler and the *Garbage Collector* (GC), in the reduction of the energy consumption of hosted applications. More formally, we formulate 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?*

By answering these questions, we envision supporting application developers and administrators in the configuration of their production environment by substantially reducing the energy consumption of hosted services at large. We also hope that our results will encourage the JVM developers to keep investing in the integration of further optimizations that can benefit a large population of software services. This paper comes with a set of contributions that can be summarized as:

(a) Reporting on the energy-efficiency of a large panel of JVM when running acknowledged benchmarks,

(b) Identifying and assessing the key JVM settings that can influence the energy consumption of a software service,

(c) Sharing guidelines and prerequisites that will help in configuring the most energy-efficiency environment before deployment,

(d) Providing a JVM benchmarking environment to evaluate the energy-efficiency of upcoming JVM distributions and their settings,

(e) Delivering an open-source tool, named J-Referral, to recommend the most energy efficient JVM distribution among 85 JVMs/versions and hundreds of configurations for Java applications.

The remainder of this paper is organized as follows. Section 6.3 introduces the experimental protocol and methodology (hardware, projects, tools, and methodology) we adopted in this study. Section 6.4 analyzes the results of our experiments on the energy consumption of the different JVM configurations. Section 6.5 discusses the related works to reduce the energy consumption of Java-based Software Services. Finally, Section 6.6 covers our conclusions.

6.2 The Java Virtual Machine

Java was originally developed by James Gosling at Sun Microsystems and released in 1995, before being acquired by Oracle in 2010. One key design goal of Java is portability, which means that Java applications must run similarly on any combination of hardware and operating system with adequate runtime support. This is achieved by compiling the Java language code to an intermediate representation, called Java *bytecode*, instead of machine code. Java bytecode instructions are analogous to machine code, but executed by a *Java Virtual Machine* (JVM), which is specific to the host machine. For example, Oracle keeps offering the HotSpot JVM, while the official reference implementation is now the OpenJDK JVM—a free and open-source software used by most developers.

Programs written in Java have a reputation for being slower and requiring more memory than those written in C++, but *Just-in-Time* (JIT) compilation, embedded in the JVM, delivers a boost of performance by opportunistically compiling bytecode to machine code at runtime. The JIT combines two compilers, C1 and C2 (also known as Client & Server VM), which are triggered based on the activity of the hosted application. Additionally, Java uses an automatic *garbage collector* (GC) to manage memory in the object lifecycle and recovering the memory once objects are no longer in use. Each JVM usually includes multiple GC, each designed to satisfy different requirements.

By default, HotSpot uses both C1 and C2 as tiered compilers,[1] and the *Garbage-First* (G1) GC with a maximum number of GC threads limited by available CPU resources and heap size, whose initial size is set $1/64^{th}$ of physical memory and maximum size may reach up to $1/4^{th}$ of physical memory.[2]

[1] http://www.ittc.ku.edu/~kulkarni/teaching/EECS768/19-Spring/Idhaya_Elango_JIT.pdf

[2] https://docs.oracle.com/en/java/javase/15/gctuning/ergonomics.html

At the time of writing this paper, the latest JVM version is Java 15, released in September 2020, while Java 11 is the current *Long-Term Support* (LTS) version. As Java 9, 10, 12, and 13 are no longer supported, Oracle advises developers to immediately transition to the latest version (currently Java 15), or an LTS release.

Beyond HOTSPOT, one can observe that the initial JVM design leads to numerous initiatives to improve the performances of Java applications, including new hotswapping strategies—with the *Dynamic Code Evolution Virtual Machine* (DCE VM) [**?**]—or alternative JIT—with GRAALVM.[3] This also includes alternative implementations, like IBM J9 JVM, which is currently distributed as part of the Eclipse foundation, and known as J9.[4] Given the wide diversity of distributions and related settings, this paper aims to study the impact of the features implemented by available JVM distributions on the energy consumption of the hosted Java software services.

6.3 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.

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

Energy Measurements. We used Intel RAPL as a physical power meter to analyze the energy consumption of the CPU package and the DRAM. RAPL is one of the most accurate tools to report on the global energy consumption of a processor [**? ?**]. We note that, due to CPU energy consumption variations issues [**?**], we used the same node for all our experiments. Moreover, we tried to be very careful, while running our experiments, not to fall in most common benchmarking "crimes" [**?**]. Every single experiment, therefore, reports on

[3] https://www.graalvm.org

[4] https://www.eclipse.org/openj9

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

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.[5]

Java Virtual Machines. We considered a set of 52 JVM distributions taken from 8 different providers/packagers mostly obtained from SDKMAN!,[6] as listed in Table 6.1. Depending on providers, either all the versions, majors, or LTS are made available by SDKMAN!.

Java Benchmarks. We ran our experiments across 12 Java benchmarks we picked from OpenBenchmarking.org.[7] This includes 5 acknowledged benchmarks from the DaCapo benchmark suite v. 9.12 [**?**], namely Avrora, H2, Lusearch, Sunflow and PMD, that have been widely used in previous studies and proven to be accurate for memory management and computer architecture communities [**? ?**]. It consists of open-source and real-world applications with non-trivial memory loads. Then, we also considered 7 additional benchmarks from the Renaissance benchmark suite [**? ?**], namely ALS, Dotty, Fj-kmeans, Neo4j, Philosophers, Reaction and Scrabble, which 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.

6.4 Experiments & Results

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

[6] https://sdkman.io/

[7] 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 codebase	Data structure, synchronization
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, transactions
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, Transactions
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 comprising two teapots and two glass spheres within an illuminated box	Compute-bound

6.4.1 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 6.1 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 6.1.

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

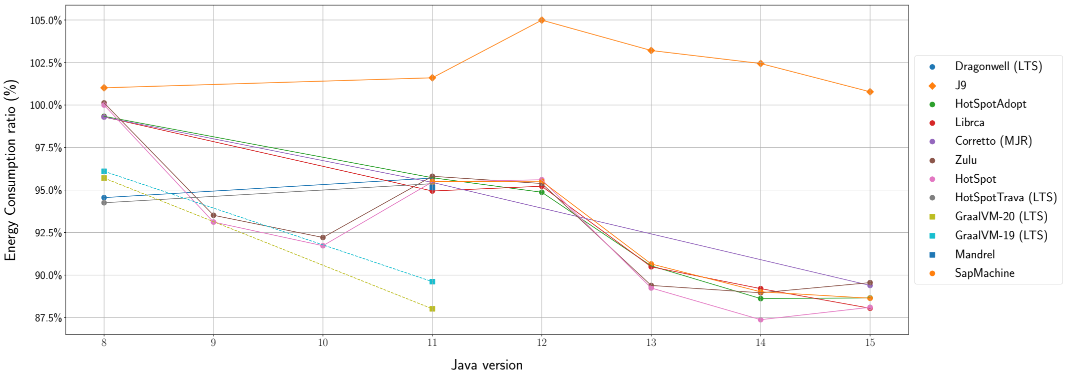


Figure 6.1: Energy consumption evolution of selected JVM distributions along versions.

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 6.1: 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 anonymous repository.[8]

Then, Figure **??** depicts the evolution of the energy consumption of the 12 benchmarks, when executed on the HotSpot JVM. Figure 6.2 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 6.1, 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

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

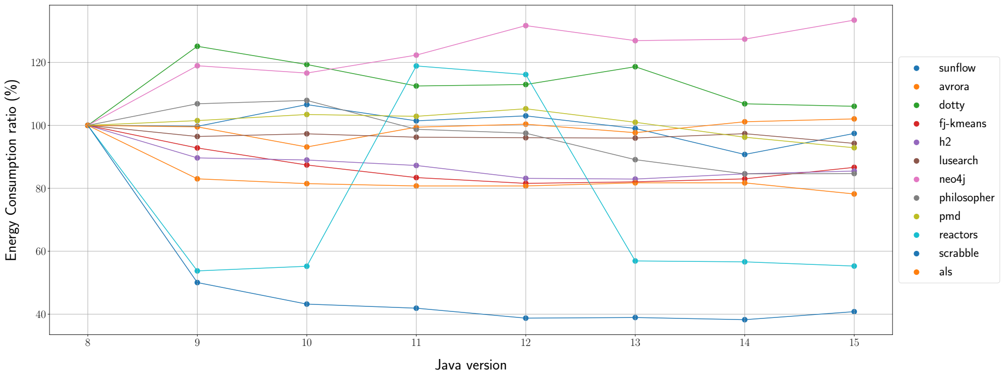


Figure 6.2: Energy consumption of the HotSpot JVM along versions.

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.[9]

Given that the wide set of distributions and versions seems to highlight 3 classes of energy behaviors, the remainder of this paper 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 6.3 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 results showed that the average power when using GRAALVM, HOTSPOT, and OPENJ9 is

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

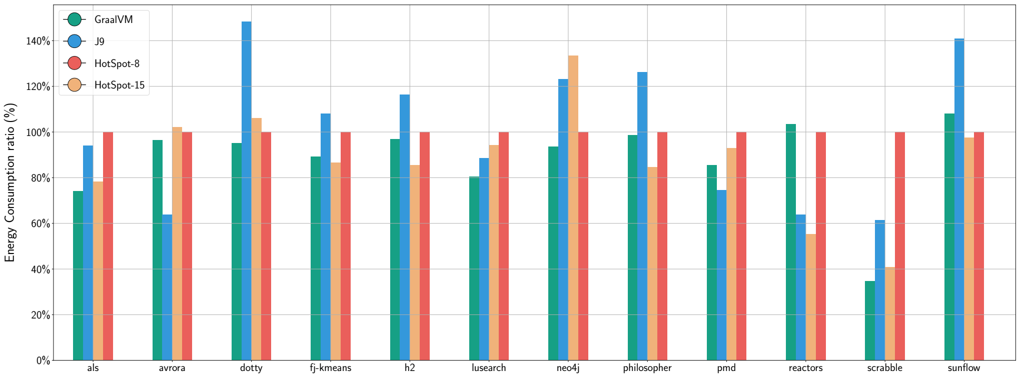


Figure 6.3: Energy consumption comparison across Java benchmarks for HotSpot, GraalVM & J9.

Table 6.3: Power per request for HotSpot, GraalVM & J9.

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

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 6.4 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 6.3 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. At the beginning of the execution, GraalVM has a slightly lower power consumption,

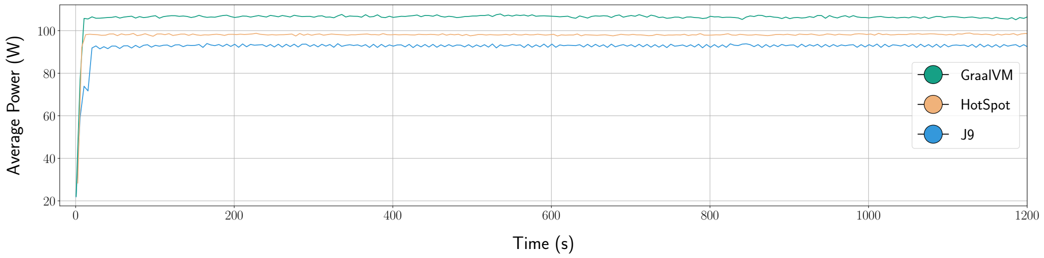


Figure 6.4: Power consumption of Scrabble as a service for HOTSPOT, GRAALVM & J9.

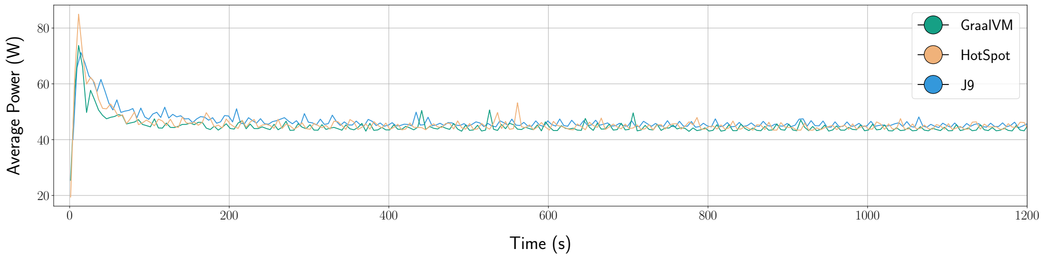


Figure 6.5: Power consumption of Dotty as a service for HOTSPOT, GRAALVM & J9.

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 6.3, 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.

6.4.2 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.

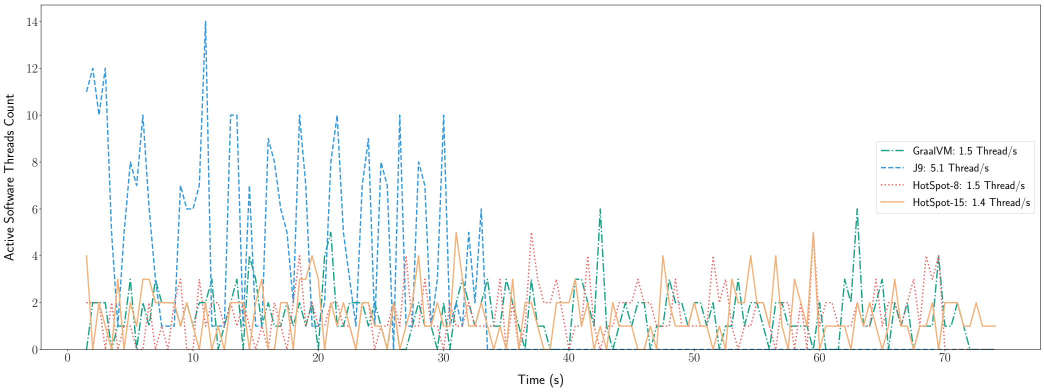
Multithreading

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.

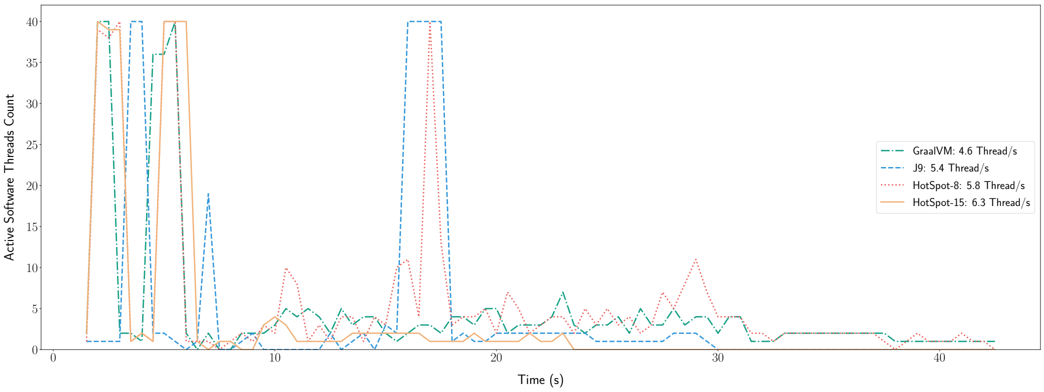
Investigating such an hypothesis requires a selection of highly parallel and CPU-intensive benchmarks, which is one of the main criteria for our benchmark selection. As no tool can accurately monitor the energy consumption at a thread level, we monitor the global power consumption and CPU utilization 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 6.3, namely Avrora and Reactors. We studied their multi-threaded behavior to optimize their energy efficiency.

Figure 6.6 delivers a closer look to the thread allocation strategies adopted by JVM. First, Figure 6.6a 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.

The second example in Figure 6.6b 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 6.3, where both JVMs consume much less energy compared to



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



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

Figure 6.6: Active threads evolution when using HOTSPOT, GRAALVM, or J9.

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).[10] 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.

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,

[10] [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		Neo4j		Pmd		Reactors		Scrabble		Sunflow	
	Default	2848	*p-values*	3861	*p-values*	2271	*p-values*	948	*p-values*	1959	*p-values*	3313	*p-values*	297	*p-values*	23452	*p-values*	452	*p-values*	335	*p-values*
GraalVM	DisableJVMCI	3099	**0.001**	4012	**0.041**	2694	**0.001**	934	**0.011**	1771	**0.005**	5086	**0.001**	353	**0.001**	25007	**0.007**	503	**0.002**	354	0.227
	Economy	4503	**0.001**	3895	0.793	3466	**0.001**	1306	**0.002**	2560	**0.001**	9525	**0.001**	270	**0.001**	30317	**0.001**	649	**0.002**	392	**0.002**
J9	*Default*	3792	*p-values*	2122	*p-values*	3515	*p-values*	1271	*p-values*	2426	*p-values*	4336	*p-values*	277	*p-values*	12705	*p-values*	734	*p-values*	476	*p-values*
	Thread 1	4157	**0.001**	2121	0.875	4749	**0.001**	1297	0.097	2597	0.066	4906	**0.001**	350	**0.001**	12800	0.713	948	**0.002**	626	**0.005**
	Thread 3	3849	0.018	2105	0.713	3574	0.104	1259	0.371	2450	0.637	4477	**0.005**	294	**0.004**	12647	0.875	795	0.021	457	0.27
	Thread 7	3843	0.041	2386	0.372	3511	0.875	1259	0.25	2424	0.637	4431	0.104	273	0.372	12600	0.875	808	0.055	463	0.372
	Count 0	8461	**0.001**	2425	**0.001**	4877	**0.001**	2289	**0.002**	3212	**0.001**	10565	**0.001**	744	**0.001**	18084	**0.001**	1476	**0.002**	922	**0.001**
	Count 1	4281	**0.001**	2150	0.431	3164	**0.001**	1841	**0.002**	2546	0.431	7166	**0.001**	272	0.128	14715	**0.001**	1005	**0.002**	514	0.052
	Count 10	3980	**0.001**	2431	0.713	3771	**0.001**	1312	**0.011**	2779	**0.003**	4979	**0.001**	299	**0.001**	12000	0.104	860	**0.005**	1182	**0.001**
	Count 100	3878	**0.007**	2141	0.713	3469	0.227	1363	0.523	2513	0.128	4547	**0.001**	262	0.031	12313	0.024	768	0.16	634	**0.004**
	Cold	6788	**0.001**	2134	0.637	4855	**0.001**	1636	**0.002**	2873	**0.001**	7250	**0.001**	275	0.372	20380	**0.001**	870	**0.005**	386	**0.001**
	Warm	4594	**0.001**	2112	0.713	4253	**0.001**	1244	0.055	2521	0.128	5305	**0.001**	411	**0.001**	13726	**0.001**	913	**0.002**	336	**0.001**
	Hot	7553	**0.001**	2310	**0.001**	12749	**0.001**	1452	**0.002**	3973	**0.001**	8979	**0.001**	857	**0.001**	36534	**0.001**	1180	**0.002**	506	0.128
	VeryHot	15113	**0.001**	3300	**0.001**	18235	**0.001**	2430	**0.002**	7205	**0.001**	19359	**0.001**	793	**0.001**	38303	**0.001**	5420	**0.002**	1692	**0.001**
	Schorching	18316	**0.001**	3541	**0.001**	21686	**0.001**	2514	**0.002**	7855	**0.001**	26409	**0.014**	808	**0.001**	43929	**0.001**	5583	**0.002**	1778	**0.001**
	Default	2997	*p-values*	4014	*p-values*	2516	*p-values*	934	*p-values*	1796	*p-values*	4787	*p-values*	323	*p-values*	11685	*p-values*	530	*p-values*	325	*p-values*
	Graal	2999	0.637	3971	0.318	2512	0.318	929	0.609	1662	**0.007**	4750	0.372	327	0.189	11548	0.523	537	0.701	338	0.564
HotSpot	Lvl 0	491443	/	14484	/	84395	/	/	/	52344	/	356287	/	1073	/	148381	/	/	/	14559	/
	Lvl 1	/	/	3731	**0.001**	3302	**0.001**	1256	**0.002**	2523	**0.001**	8304	**0.001**	222	**0.001**	22410	**0.002**	735	**0.002**	277	**0.007**
	Lvl 2	3079	**0.004**	4110	0.189	3723	**0.001**	22547	**0.002**	2840	**0.001**	19058	**0.001**	226	**0.001**	40701	**0.002**	2291	**0.002**	4131	**0.001**
	Lvl 3	16375	**0.001**	7729	**0.001**	6789	**0.001**	144914	**0.002**	4139	**0.001**	44594	**0.001**	330	**0.005**	190124	**0.002**	9070	**0.002**	10449	**0.001**
	NotTired	3254	**0.001**	3901	0.189	3110	**0.001**	912	**0.021**	1846	0.227	3844	**0.001**	933	**0.001**	11256	**0.041**	588	**0.003**	405	**0.001**

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

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

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.

Garbage Collection

Changing or tuning the GC strategy has been acknowledged to impact the JVM performances [**?**]. 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 in bold show the values that are significantly different from the default configuration, where

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

Table 6.7: Energy consumption when tuning GC settings on HotSpot, GraalVM & J9

JVM	Mode	ALS		Avrora		Dotty		H2		Neo4j		Pmd		Reactors		Scrabble		Sunflow	
	Default	2570	*p-values*	4153	*p-values*	2223	*p-values*	1870	*p-values*	5256	*p-values*	281	*p-values*	2611	*p-values*	410	*p-values*	353	*p-values*
	1Concurent	2567	0.403	4007	**0.023**	2220	1.000	1883	0.982	5368	1.000	286	0.182	2664	1.000	413	0.885	347	0.573
	1Parallel	2668	**0.012**	3904	**0.008**	2228	0.835	2022	**0.000**	5836	**0.012**	298	**0.000**	2869	0.144	561	**0.030**	317	**0.000**
	5Concurent	2570	0.676	4117	0.161	2215	0.210	1862	0.505	5259	1.000	282	0.980	2611	0.531	414	0.885	362	0.356
GraalVM	5Parallel	2561	0.676	3863	**0.012**	2237	1.000	1910	0.103	5223	0.403	282	0.538	2682	0.531	424	0.112	353	0.758
	DisableExplicitGC	2559	0.210	3911	**0.003**	2215	1.000	1978	**0.018**	5106	0.210	281	0.758	2704	0.676	400	0.312	332	**0.036**
	ParallelCG	2720	**0.012**	4016	0.206	2237	0.531	1945	**0.000**	13172	**0.037**	282	0.878	2267	**0.022**	545	**0.030**	329	**0.003**
	ParallelOldGC	2715	**0.012**	4032	0.103	2221	1.000	1925	**0.002**	13362	/	282	0.918	2514	**0.012**	535	**0.030**	329	**0.008**
	Default	3371	*p-values*	2243	*p-values*	3237	*p-values*	2107	*p-values*	6277	*p-values*	232	*p-values*	1644	*p-values*	589	*p-values*	510	*p-values*
	Balanced	9012	**0.012**	2232	0.597	3429	**0.012**	2247	**0.002**	8853	**0.012**	235	0.412	1902	**0.020**	661	0.061	519	0.505
	ConcurrentScavenge	3487	**0.012**	2270	0.280	3388	**0.012**	2319	**0.001**	6857	**0.012**	233	0.878	1705	0.903	639	0.194	546	**0.018**
	Metronome	2098	**0.012**	2265	0.505	3815	**0.012**	2717	**0.000**	12103	**0.012**	239	**0.022**	2089	**0.020**	758	**0.030**	422	**0.000**
J9	Nogc	3454	**0.022**	2239	0.872	3259	0.144	2207	0.031	61781	**0.012**	227	0.151	1505	**0.066**	711	**0.030**	499	0.720
	Optavgpause	3601	**0.012**	2431	0.370	3425	**0.012**	2169	0.297	7495	**0.012**	253	**0.000**	1772	0.391	1089	**0.030**	478	**0.046**
	Optthruput	3357	1.000	2432	0.241	3178	0.403	2194	0.139	6324	0.835	232	0.878	1554	0.111	640	0.194	429	**0.000**
	ScvNoAdaptiveTenure	3494	**0.012**	2253	0.800	3248	0.835	2161	0.103	8442	**0.012**	228	0.137	1908	**0.020**	618	0.665	528	0.218
	Default	2765	*p-values*	4115	*p-values*	2492	*p-values*	1673	*p-values*	8152	*p-values*	316	*p-values*	1546	*p-values*	484	*p-values*	347	*p-values*
	1Concurent	2775	0.060	4137	0.346	2493	0.676	1675	0.918	8062	0.531	316	0.383	1533	0.665	478	0.470	334	**0.218**
	1Parallel	2863	**0.012**	4142	0.800	2526	**0.037**	1853	**0.001**	8270	0.676	334	**0.000**	1747	**0.030**	592	**0.030**	320	**0.002**
	5Concurent	2758	0.676	4091	0.872	2485	0.296	1681	0.608	8087	0.835	314	0.330	1497	0.665	469	**0.030**	336	0.259
	5Parallel	2767	0.144	4176	0.077	2473	0.060	1654	0.720	8046	0.835	316	0.573	1546	0.470	489	0.470	342	0.573
HotSpot	DisableExplicitGC	2734	**0.012**	4062	0.448	2483	0.835	1702	0.248	7710	**0.037**	312	0.200	1545	0.470	470	0.061	325	**0.014**
	ParallelCG	2653	**0.012**	4064	0.629	2356	**0.012**	1602	**0.008**	8953	0.060	300	**0.000**	1476	0.885	579	**0.030**	336	0.081
	ParallelOldGC	2764	0.531	4070	0.872	2525	0.802	1675	0.959	7963	0.403	314	0.720	1582	0.194	475	0.470	333	0.151
	SerialGC	2593	**0.012**	4083	0.395	2378	**0.012**	1620	**0.046**	5745	**0.012**	307	**0.002**	1672	0.061	601	**0.030**	352	0.473

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.

J-Referral

To help developers and practitioners choosing an energy efficient JVM distribution and configuration, we propose a tool named J-Referral. This tool takes a Java application as an input and a launch script to tune the JVM configuration. The recommendation is is based on the 85 JVM distributions and versions made available by SDKMAN!.[11] Beyond the JVM distribution and version recommendation, J-Referral automatically explores many GC and JIT options for each distribution. Hundreds of combinations are thus compared to recommend an energy efficient JVM distribution and the associated configuration parameters.

Table 6.8 illustrates an example of the final report of J-Referral. The tool was tested for 2 real Java projects: Zip4J and[12] and K-nucleotide.[13]. Zip4J runs a large file compression,

[11] https://sdkman.io/

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

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

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

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.[14]

6.5 Related Work

Practitioners keep looking for development tools and methods to deliver software that meets both performance and quality requirements. In this context, energy consumption is becoming of growing importance when deploying software services in the cloud, which generally promotes a *pay-as-you-go* pricing model, thus challenging the resource-efficiency of deployed services. So far, software engineering tackles this optimization challenge along three axes: 1. programming languages, 2. source code, 3. execution platforms

In this section, we review the state of the art of studies related to JVM energy consumption.

At code level. Many works investigated software energy consumption efficiency through source code changes and optimizations. For example, [19, 8] studied the effect of Java collections on energy consumption, with regards to the collection size and/or the most executed tasks on the collection (insertion, removal, search). In [20], Hasan *et al.* compared the energy consumption of several Java data structures, analyzing the bytecode using the Wala framework[15] and assessing the evolution of the energy consumption in different scenarios (insertion at the beginning, iteration, etc.). They also used some automated replacement of `LinkedList` and `ArrayList` to simulate best- and worst-case energy consumption scenarios

[14] https://anonymous.4open.science/r/jreferral/Readme.md

[15] http://wala.sourceforge.net/wiki/index.php

on real production applications. Their study showed that using inappropriate collection can cause an energy consumption inefficiency of 300% for the worst-case scenario.

SEEDS and SEEDS-API is a fully automated framework, proposed in [?], to analyze source code (at bytecode level for Java) and auto-tunes apps to have reduce their energy consumption, with a focus on Java collection tuning. The authors reported on an improvement up to 17%. In [17], the same authors presented SPELL, the energy leaks detector tool. The tool uses JRAPL [? 19] to detect energy-inefficient code fragments by using a statistical spectrum-based energy red spots localization. Their evaluation reported up to 18% energy savings on Java applications.

Some researchers tried to highlight the impact that has some atomic code changes on the energy consumption. Many papers studied the impact of code refactoring on energy consumption [?]. Other papers investigated the energy consumption of Java primitive types, operations on strings, usage of exceptions, loops, and arrays [14].

At JVM level. In [18], Pereira *et al.* investigated several programming languages, including JVM-based languages like Java and Scala, and compared them on different dimensions (energy, memory, and time) using CLBG benchmarks. Their observation reported that compiled languages are more energy-efficient than interpreted ones, overall. More advanced results expose how the selected languages may satisfy one, two, or three dimensions of the equation. The authors of [?] conducted a performance analysis and comparison between HOTSPOT and J9. They claimed in their results that the relative performance of HOTSPOT ranges from 44% to 289% of J9, while the dynamic power consumption varies from 2.7W to 7.2W using the SPECjvm2008 benchmarks.

The difference between HotSpot and J9 was also reported in other studies. Chiba *et al.* [?] evaluated the effect that could have those 2 JVM platforms on the performance of a combination of big data query engines (SPARK and TEZ) using TPC-DS benchmark. They reported on a 3-fold drawback that can exhibit one JVM compared to the other.

On another note, the authors of [15] attempted to assess and assign an energy cost to atomic bytecode instructions together with a constant overhead of using the JVM. Their claims are rather surprising, as the energy cost should be impacted by the execution environment, workload, core frequencies, and can hardly be fixed. A similar idea was used in [?] to design a model for JVM-based software energy consumption, using a bytecode-level model. The authors described their tool, named OPACITOR, as being deterministic, accurate, and robust to the surrounding noise. However, they disable the JIT in their experiments to maintain the deterministic behavior of their tool, which does not reflect a real software execution given all the optimizations that the JVM triggers to optimize the performances.

6.6 Conclusion

This paper reports on an empirical investigation of the key differences in energy consumption that some of the most famous and supported JVM platforms can exhibit, in addition to the key settings that can impact this energy consumption positively or negatively. During our experiments, we considered a total of 12 well-known and diversified-purposes Java benchmarks together with a total of 52 JVMs, including many versions of 11 different distributions. 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, 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

[1] 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 - ICS '16*, pages 1–12, Istanbul, Turkey. ACM Press.

[2] Borkar, S. (2005). Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16.

[3] Chasapis, D., Schulz, M., Casas, M., Ayguadé, E., Valero, M., Moretó, M., and Labarta, J. (2016). Runtime-Guided Mitigation of Manufacturing Variability in Power-Constrained Multi-Socket NUMA Nodes. In *Proceedings of the 2016 International Conference on Supercomputing - ICS '16*, pages 1–12, Istanbul, Turkey. ACM Press.

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

[5] Echtler, F. and Häußler, M. (2018). Open source, open science, and the replication crisis in hci. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–8.

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

[7] El Mehdi Diouri, M., Gluck, O., Lefevre, L., and Mignot, J.-C. (2013). Your cluster is not power homogeneous: Take care when designing green schedulers! In *2013 International Green Computing Conference Proceedings*, pages 1–10, Arlington, VA, USA. IEEE.

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

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

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

[11] Howe, B. (2012). Virtual Appliances, Cloud Computing, and Reproducible Research. *Computing in Science Engineering*, 14(4):36–41. Conference Name: Computing in Science Engineering.

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

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

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

[15] Lafond, S. and Lilius, J. (2006). An Energy Consumption Model for an Embedded Java Virtual Machine. In Grass, W., Sick, B., and Waldschmidt, K., editors, *Architecture of Computing Systems - ARCS 2006*, volume 3894, pages 311–325. Springer Berlin Heidelberg, Berlin, Heidelberg.

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

[17] Pereira, R., Carcao, T., Couto, M., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017a). Helping Programmers Improve the Energy Efficiency of Source Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 238–240, Buenos Aires, Argentina. IEEE.

[18] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017b). Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA. ACM.

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

[20] Samir Hasan, Rachary King, and Munawar Hafiz (2016). Energy Profiles of Java Collections Classes.

[21] The shift Project (2019). Lean ICT - Towards Digital Sobriety. Technical report.

[22] 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):1396–1402.

[23] Varsamopoulos, G., Banerjee, A., and Gupta, S. K. S. (2009). Energy Efficiency of Thermal-Aware Job Scheduling Algorithms under Various Cooling Models. In Ranka, S., Aluru, S., Buyya, R., Chung, Y.-C., Dua, S., Grama, A., Gupta, S. K. S., Kumar, R., and Phoha, V. V., editors, *Contemporary Computing*, volume 40, pages 568–580. Springer Berlin Heidelberg, Berlin, Heidelberg.

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

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