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 August 2022

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

Declaration

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

Mohammed Chakib Belgaid
August 2022

Acknowledgements

And I would like to acknowledge ...

Abstract

This is where you write your abstract …

Contents

List of Figures

List of Tables

Chapter 1

The Impact of Python Runtime on Energy Consumption

1.1 Introduction

Dynamic programming languages, except Perl, have surpassed compiled programming languages in terms of popularity among software system developers over the past decade (cf. Figure 1.1). However, it remains unclear whether this category of dynamic programming languages can truly compete with compiled ones in terms of power consumption.

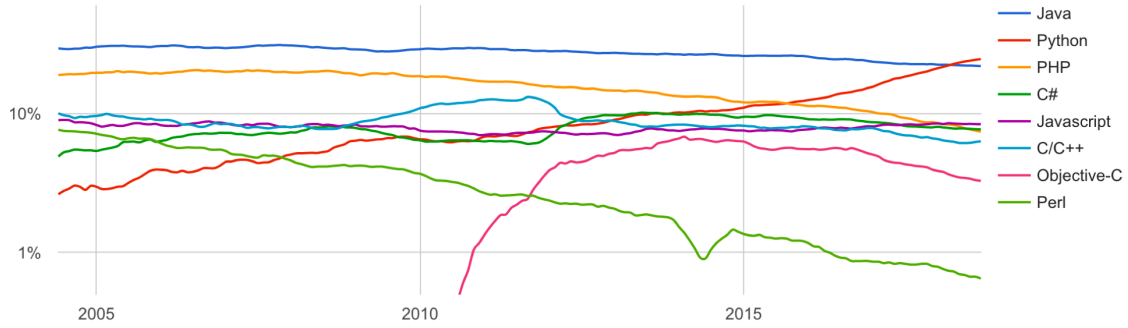


Figure 1.1: PYPL Popularity of Programming Languages [noa].

In particular, Noureddine *et al.* [12] in 2012, and then Pereira *et al.* [16] in 2017, conducted empirical power measurements on this topic, and both concluded that compiled programming languages overcome dynamic ones when it comes to power consumption. According to their experiments, an interpreted programming language, like Python, can impose up to a $7,588\,\%$ energy overhead compared to C [16] (cf. Figure 1.2). In this chapter, we explore the oblivious optimizations that can be applied to Python legacy applications to

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

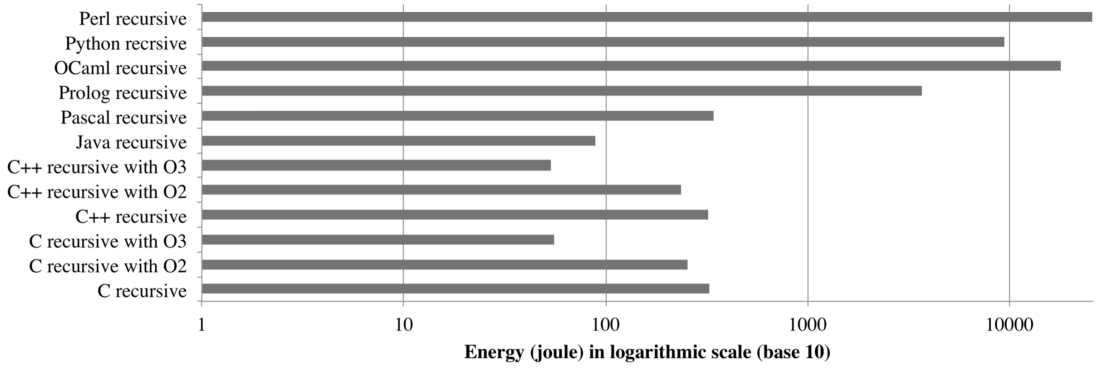


Figure 1.2: Energy consumption of a recursive implementation of Tower of Hanoi program in different languages [12]

1.2 Motivation

1.2.1 Python Popularity

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

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

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

1.2.2 Python Gluttony

According to [17] and [12], Python tends to be one more energy hungry programming language. As one can notice in Figure 1.2, Python consumes 30 times more than C or C++. The benchmark was done with an implementation of the Tower of Hanoi[2] of 30 disks.

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

As shown in Table 1.1, one can observe that, for most of the applications taken for the *Computer Language Benchmark Game* (CLBG), Python takes more time to execute—the only case that he was not the worst one was in the benchmark regx-redux where he beat Go—and in some cases the gap was huge, such as in n-body where Python took around 100 times more than C++.[3]

Table 1.1: Comparison of CLBG execution times (in seconds) depending on programming languages.

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

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

1.2.3 Python Limits

To reduce the energy consumption of Python, we started by targeting the main usage of this programming language, which is revealed to be data science and web development. Figure 1.3 illustrates a study published by the JetBrain company on Python developers.[4]. In this survey, multiple answers were accepted. 57% of the respondents reported that they use Python for data science, 51% said they are using it for web development, and around 40% are using it for system administration.

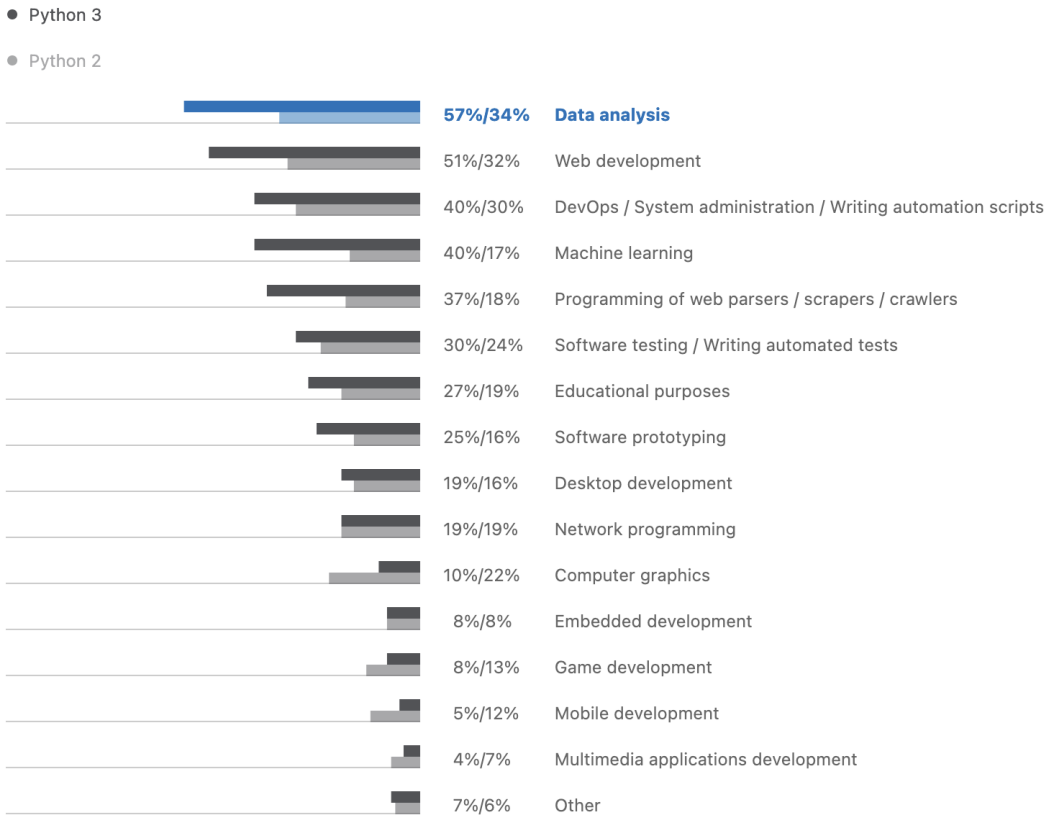


Figure 1.3: Use cases of Python (source: JetBrain).

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

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

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

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

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

In this chapter, we report on 4 case studies that intend to answer these research questions. First, we study the energy behavior of Python in two application contexts: web applications (cf. Section 1.3) and machine learning. Then, we dive deeper into the energy consumption of Python core structures, before concluding with the impact of the Python interpreter on energy consumption.

1.3 Green Web Development

We start by studying the impact of the *Object Relational Mapping* (ORM) [14] on energy consumption. The idea behind this use case is to observe if the choice of the database and the ORM impacts the energy consumption and the performance of the website. We, therefore, create a Django website to analyze the cost of a single request using various mechanisms to retrieve data from the database.

We considered using two different databases, POSTGRESQL and SQLITE3, that store the same records, and three different ways to fetch the records.

1. Vanilla relies only on the ORM to retrieve the data,
2. Prefetch queries the data before being requested,
3. Optimized leverages SQL without passing by the ORM.

As one can observe in Figure 1.4, the strategy to query stored data has a huge impact on the energy consumption. As the Vanilla strategy can consume up to 10× more energy than the Optimized one. Conversely, the choice of the database does not exhibit a key impact on the total energy despite their different behavior regarding the execution time and the average power. This can be useful to support developers in choosing which database engine they can adopt, based on the number of expected requests and the targeted performance.

Another interesting observation is the impact of the interpreter, as Figure 1.4 highlights. For example, using the Pypy interpreter reduces the energy consumption, even when we adopting the Vanilla strategy.

Figure 1.8 reports on the energy consumption of Python applications when introducing parallelism. Python applications are single-threaded systems constrained by the *global lock system* (GLS). However, due to the increase of cores/threads per CPU, many Python libraries started to take advantage of this hardware feature by allowing concurrent execution of Python

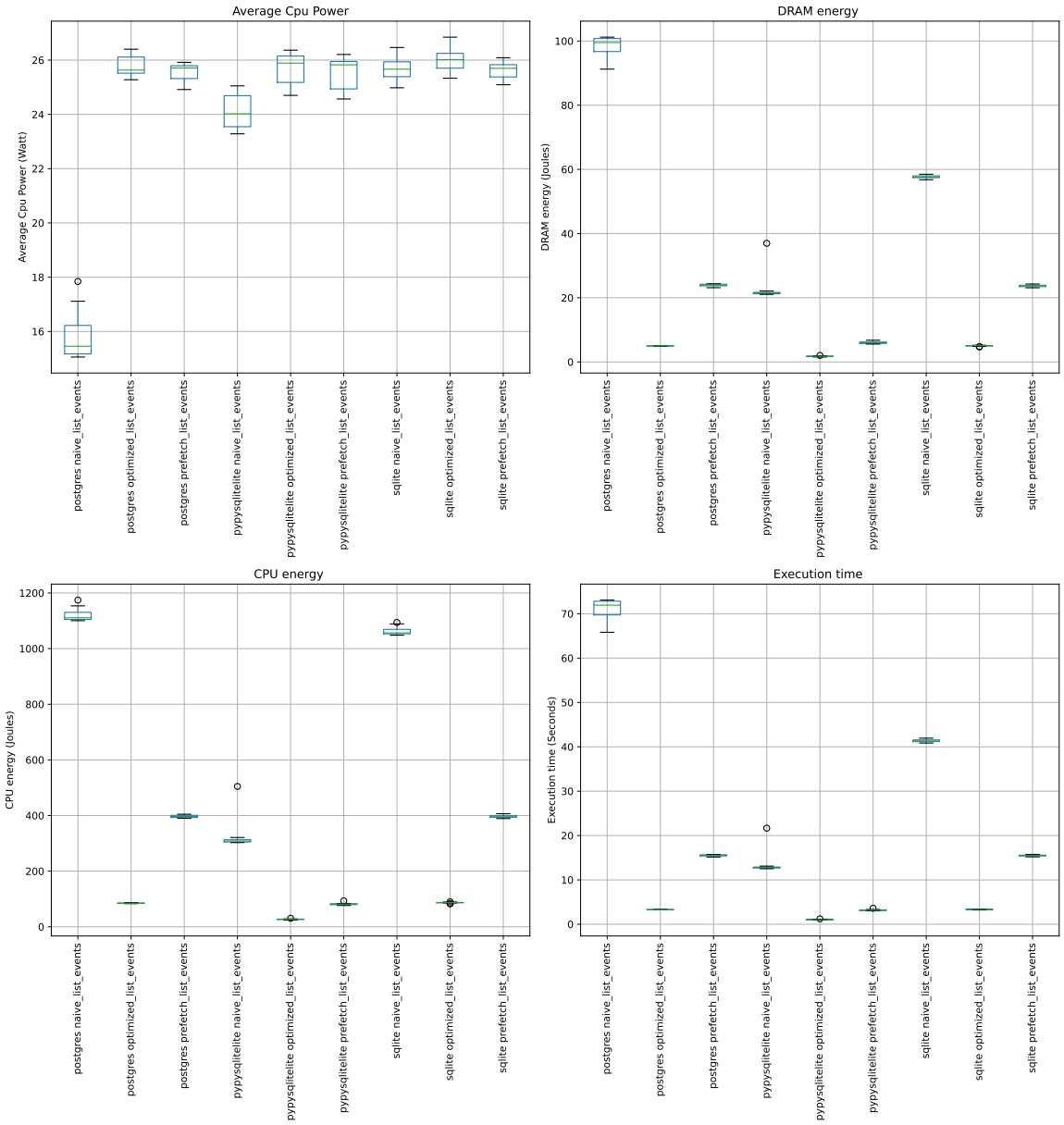


Figure 1.4: Energy behavior resulting from data access strategies.

processes. For example, the multiprocessing library[5] spawns sub-processes to increase the degree of concurrency of Python applications. As one can see in Figure 1.8, the energy consumption is correlated with the number of threads until reaching the limit of physical cores, when concurrent processes start to compete for the CPU.

Before reaching the limit of physical cores, we also observe that the scheduler of the operating system tends to favor the execution of processes on the same physical core, by

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

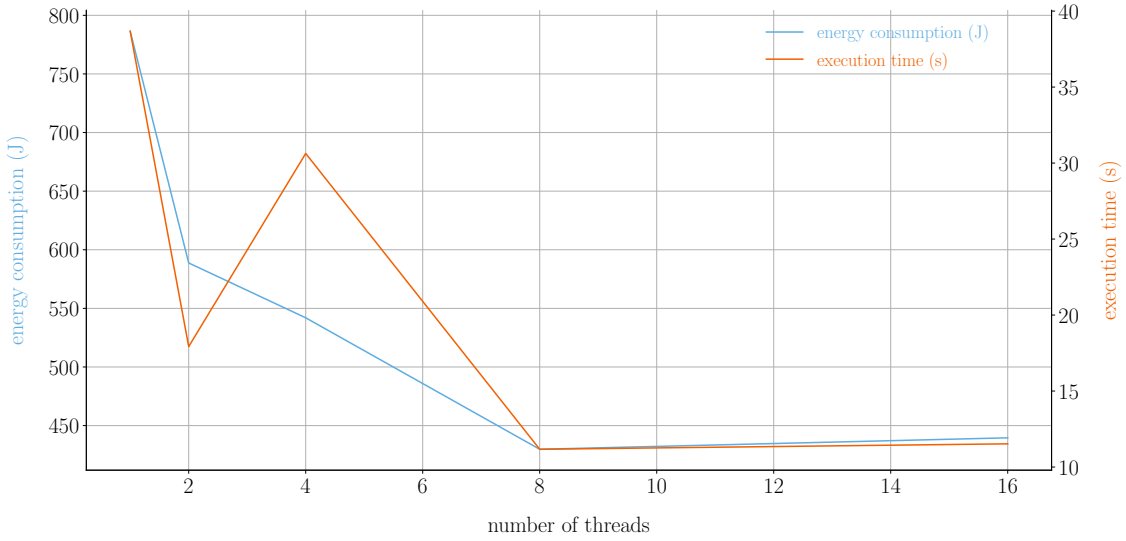


Figure 1.5: Energy consumption of Python multiprocessing depending on the number of exploited threads.

taking advantage of the hyper-thread feature. While this strategy aims to save energy by leveraging the ACPI P-states and C-states of unallocated cores, this leads to increase execution times.

1.3.1 Python Insights

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

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

As one can see in Figure 1.6, the type of the data has no impact on the energy consumption. However, the way one iterates over the collection has a huge factor. Interestingly, the `for`

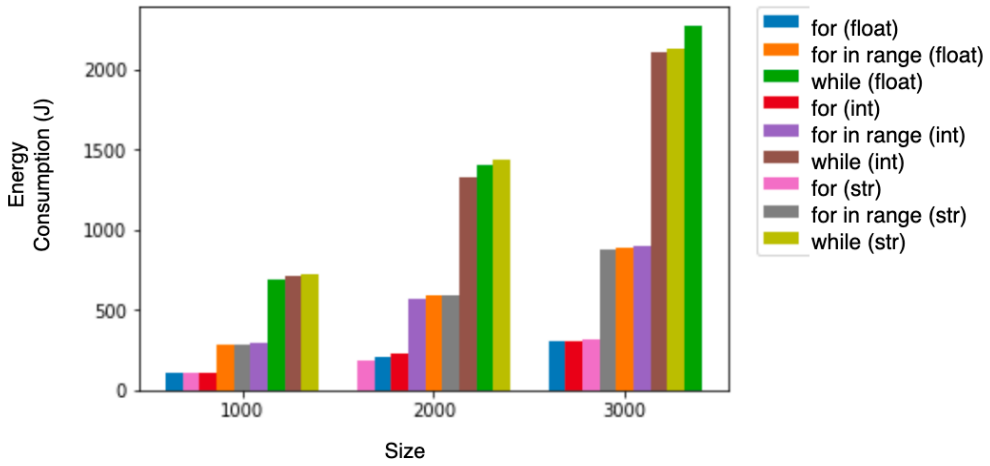


Figure 1.6: Comparison of the energy consumption of different Python loops.

`in range` loop was by far the optimal one, followed by the regular `for in collection`, and the `while` part was the last one with an overhead of 400% compared to the first option.

The reason behind such a behavior is mainly related to how the Python interpreter is implemented. To reduce the latency of Python applications, most of the built-in functions and operations are written in C, and the same goes for the function `range`. Furthermore, the function `len` has a complexity of $\mathcal{O}(1)$ as it is based on the function `Py_SIZE` of C, which stores the length in a field for the object. Therefore, the `for in range` is creating a new iterator that has the same length as the first one and, for each iteration, requires a second access (`l[i]`) instead of one—explaining the doubled time. The `while` is even slower due to the implicit increment of the variable, which causes an extra operation during the loop. To confirm this hypothesis, we tried to construct a new list by editing the elements of the previous one (cf. Figure 1.7). And, as predicted, the built-in methods are the most energy-saving ones, while the customize `while` loop is the heaviest.

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

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

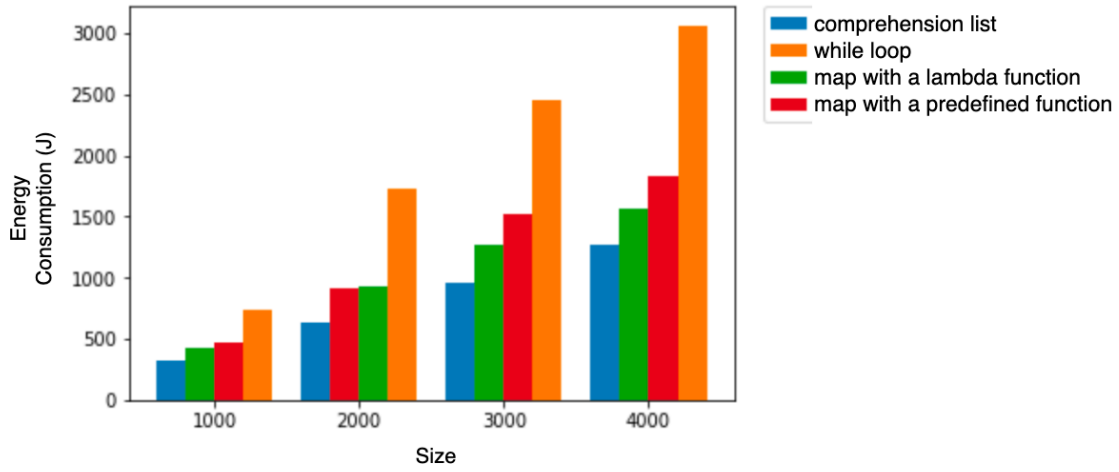


Figure 1.7: Comparison of the energy consumption of different methods to convert a list.

1.3.2 Python & Multiprocessing

Figure 1.8 compares the behaviour of python programs when we try to introduce the parallelism. As we know python is a single threaded program thx to ghe GLS (global lock system) howerver due to the increase of the number of cores /threads per cpu it many libraries stared to take advanteage of this feature so most of them they will try to simulate the multithreading by using multiple instances pf the processor

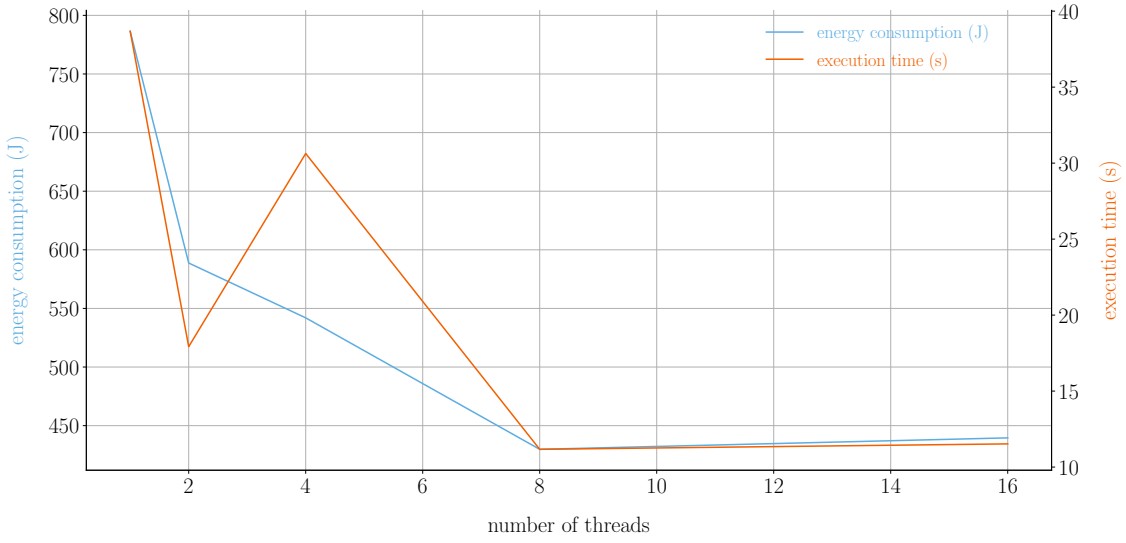


Figure 1.8: energy behaviour based on multiprocessing

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

1.3.3 Python & Machine Learning

Machine learning is becoming an integral part of our daily lives, growing more potent and energy-hungry each year.

As machine learning can have a significant impact on climate change, it is vital to investigate mitigation techniques.

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

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

Input Workload

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

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

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

candidates

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

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

Key Performance Metrics

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

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

1.4 Results & Findings

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

Figure 1.10 depicts how the curve steepens as training progresses. For example, training to 90% accuracy requires three times the energy required for training to 80% accuracy.

Conclusion We discovered that when a model's accuracy improves, so does the energy required for a subsequent accuracy gain. This begs the question of when we should discontinue training. Is a 10% increase in accuracy worth it if we have to spend three times the energy? Even while utilizing a GPU consumes more total power, the training time is significantly reduced. As a result, the utilization of a GPU is required to lower the energy consumption

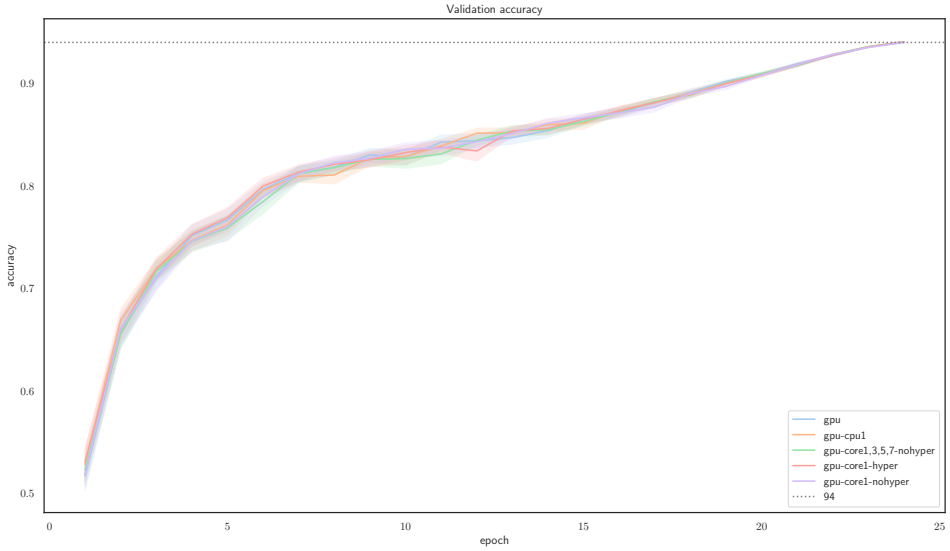


Figure 1.9: accuracy based on epoch

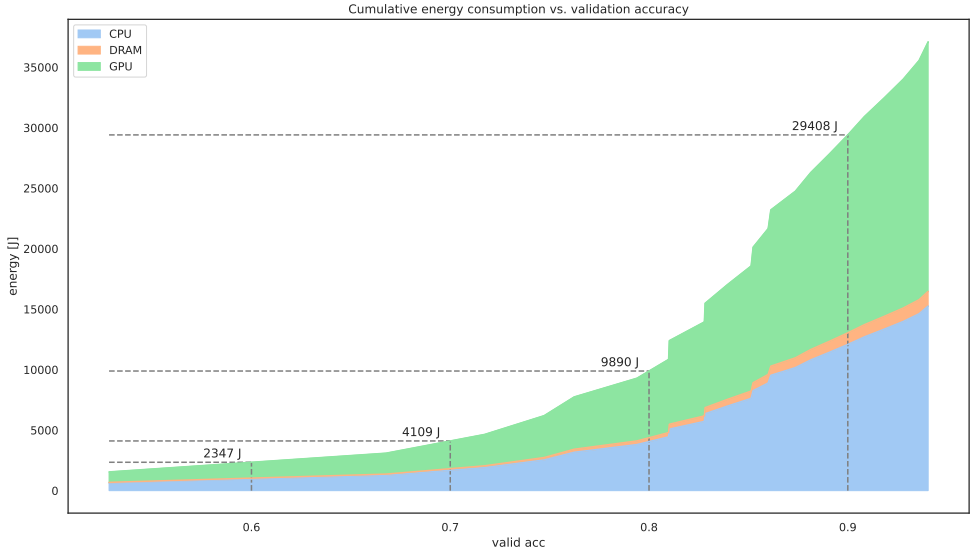


Figure 1.10: cumulative energy consumption vs accuracy

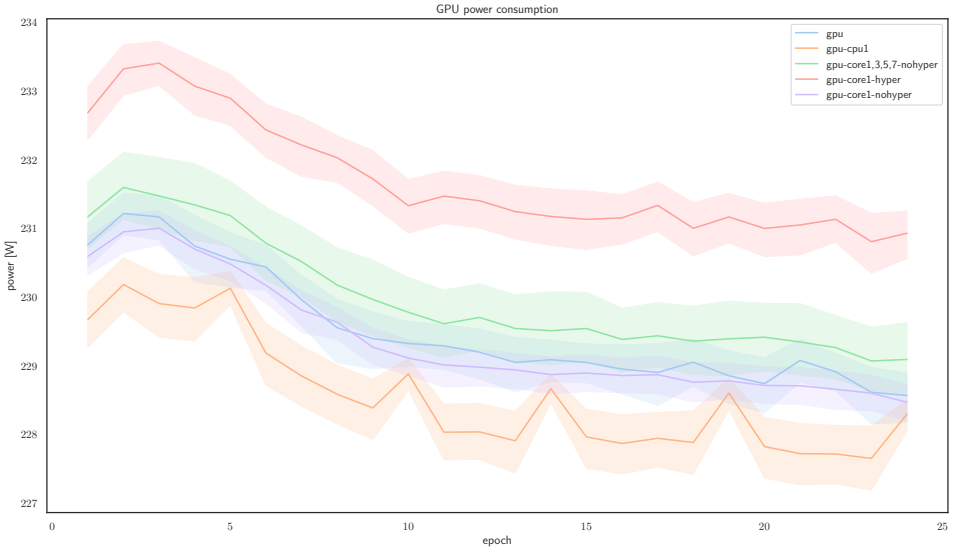


Figure 1.11: evolution of average gpu power based on epoch

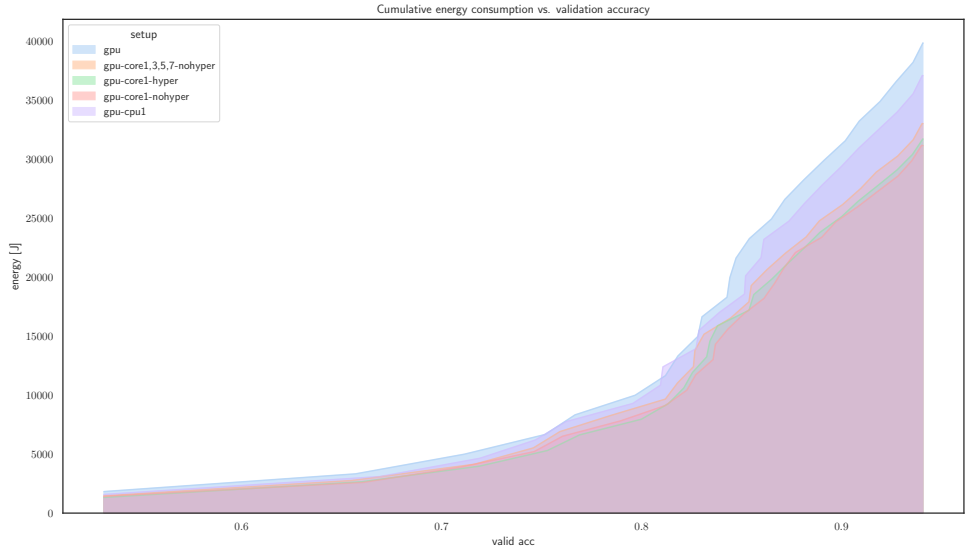


Figure 1.12: cumulative energy of fast10 benchmark within different configurations

of the training. Some experiments revealed unusual behavior that was not studied during this internship. These could be examined in future studies and perhaps used to achieve more energy savings. The execution time did not depend on the number of cores employed in several studies. This parameter had a significant impact on training time in others. As a result, the best core configuration will be determined by the script. The next steps in this area could include exploring the reasons for these discrepancies and possibly creating a script that can automatically find the ideal core configuration for a specific script.

1.5 Python Interpreters

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

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

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

1. **CPython**:[6] This Python interpreter, written in C, is the reference interpreter of Python. CPython compiles the source code into byte-code and then interprets it. The CPython project supports both versions of Python 2 and 3;

2. **PyPy**:[7] An alternative implementation of the Python interpreter. It is written using *RPython* to use the JIT. It compiles the most used portions of the Python code into a binary code for better performance. To benefit from these optimizations, the program

[6]https://www.python.org/
[7]http://pypy.org

has to be executed for at least for few seconds so the JIT has enough time to warm up, the JIT optimization are only applied to the code written by the developer and not to external libraries;

3. **Cython**:[8] A static compiler for Python. It translates the Python code into C, and then compiles it using a C compiler. It also supports an extended version of the Python language that allows programmers to call *C functions*, declare *C types* and use static types, which will help the translation of Python objects into native types, such as integers, float. This often means better performances, since native C libraries are almost all the time faster than the Python written once [16];

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

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

6. **IronPython**:[13] A .Net-based Python interpretation platform written in C# that is used with the .Net virtual machine or Mono. It benefits from all the optimizations of .Net virtual machines, such as the JIT and garbage collector mechanisms;

7. **GraalPython**:[14] A Python interpreter that is based on GraalVM[15] (a universal virtual machine developed by oracle for running applications written in different programming languages). For the time being, it only supports Python 3 and it is still in the experimental stage;

8. **Jython**:[16] An implementation of Python programming language written in Java for the *Java Virtual Machine* (JVM). Similar to IronPython and GraalPython, it leverages the optimization mechanisms provided by the JVM to enhance the Python performances;

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

9. **MicroPython**:[17] A lightweight Python version dedicated to embedded systems and micro-controllers;
10. **Nuitka**:[18] A Python compiler written in Python that generates a binary executable from Python code. It translates the Python code into a C program that is then compiled into a binary executable;
11. **Numba**:[19] A library that includes JIT compiler to enhance the performances of Python functions using the industry-standard LLVM compiler library;
12. **Shedskin**:[20] A static transpiler that translates implicitly statically typed python into C++ code;
13. **Hope** [2]: A Python library that aims to introduce JIT compiler into the Python code;
14. **Parakeet** [**?**]: A runtime accelerator for an array-oriented subset of Python;
15. **Stackless Python**:[21] An interpreter that focuses on enhancing multi-threading programming;
16. **Pyjion**:[22] A JIT API for CPython, same purpose as Parakeet and Hope;
17. **Pyston**:[23] A performance-oriented Python implementation built using LLVM and modern JIT techniques. The project is funded by Dropbox;
18. **Grumpy**:[24] A source-to-source transpiler that translates the Python code into Go before being compiled to a binary executable. It also offers an interpreter, called *grumprun*, which can directly execute the Python code. Unfortunately, we cannot use it because the project is already outdated (last commit is in 2017) and it has a lot of limitations in terms of supporting the Python language, such as some built-in functions and standard libraries;
19. **Psyco**:[25] A JIT compiler for Python;
20. **Unladen Swallow**:[26] An attempt to (use) LLVM as a JIT compiler for CPython.

[17] http://micropython.org
[18] http://nuitka.net/pages/overview.html
[19] https://numba.pydata.org
[20] https://github.com/shedskin/shedskin
[21] https://github.com/stackless-dev/stackless/wiki
[22] https://github.com/microsoft/pyjion
[23] https://blog.pyston.org
[24] https://github.com/google/grumpy
[25] http://psyco.sourceforge.net
[26] https://unladen-swallow.readthedocs.io/en/latest/

1.5.1 Runtime Classification

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

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

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

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

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

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

We should explain the classification of these runtimes

There are other implementations that we did not consider because either the project aborted many years ago or it has very limited support for Python features. After the collection of those implementations, we filtered them. To keep only the versions that are still maintained and support most Python features. and we classified them into 3 categories depending on their integration with the python code. In Table 1.3, we describe the implementations that we kept and the version of each implementation and its category.

1.6 experimetal protocole

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

Table 1.2: Classification of Python implementations

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

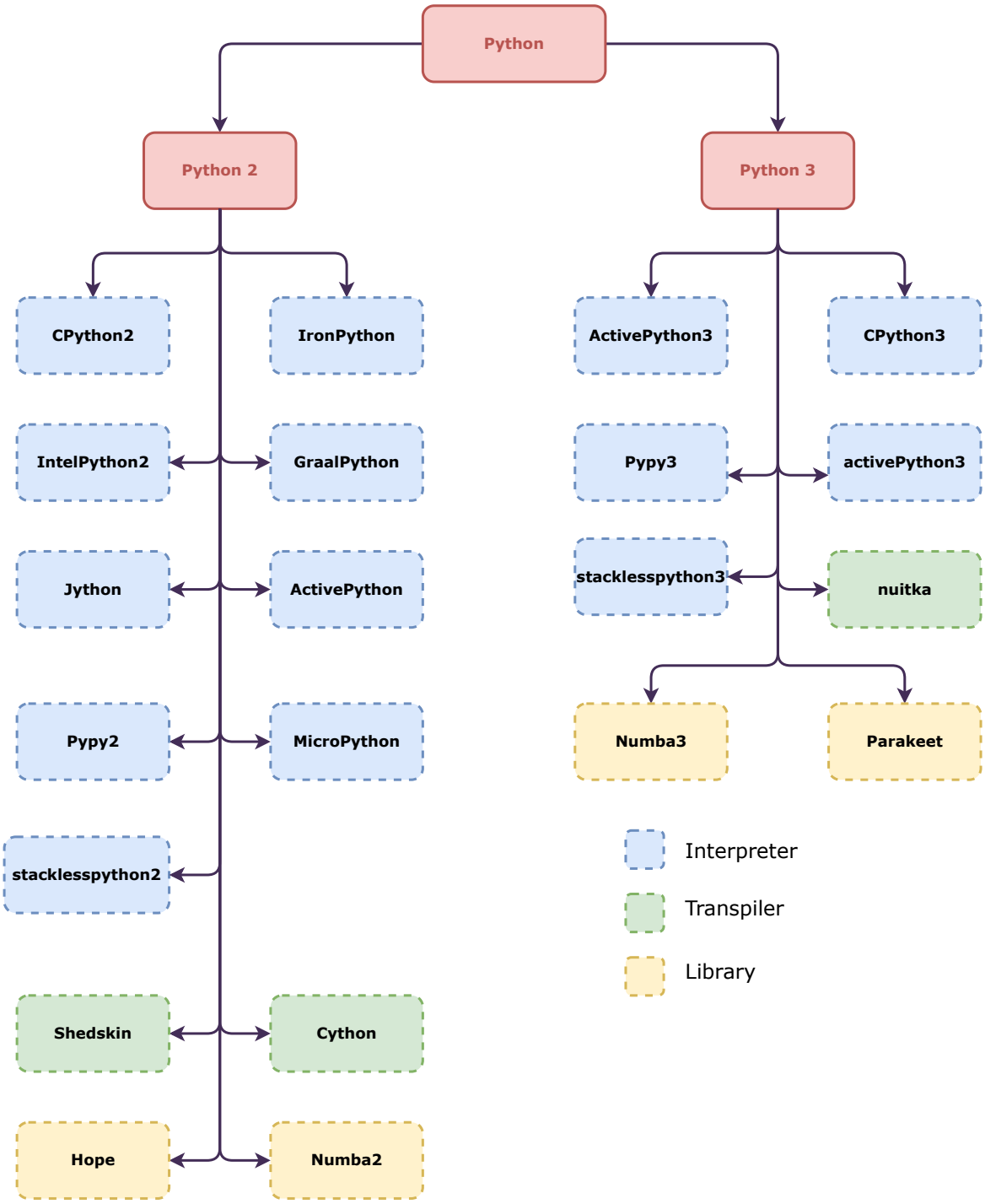


Figure 1.13: Python interpeters

Table 1.3: Classification of Python implementations

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

1.6.1 measurement context

Hardware settings all our tests have been executed in a Dell PowerEdge C6420 server machine. A summary of its hardware is listed in table 1.4. The machine is equiped with a minimal version of Debian 9 (4.9.0 kernel version) where we install Docker (version 18.09.5).

CPU	Intel Xeon Gold 6130 (Skylake, 2.10GHz, 2 CPUs/node, 16 cores/CPU)
Memory	192 GiB
Storage	240 GB SSD SATA Samsung MZ7KM240HMHQ0D3
	480 GB SSD SATA Samsung MZ7KM480HMHQ0D3
	4.0 TB HDD SATA Seagate
Network	eth0/enp24s0f0, Ethernet, configured rate: 10 Gbps, model: Intel Ethernet Controller X710 for 1
	ib0, Omni-Path, configured rate: 100 Gbps, model: Intel Omni-Path HFI Silicon 100 Series [disc

Table 1.4: Testing Machine Configuration

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

1.6.2 Metrics

Our focus will be mainly on CPU energy consumption because it is ten folds more than the DRAM one, since it is finit job benchmarking, time is highly correlated within the energy, and it will be only useful to explain certain energytical behaviour so we wont put a lot of focus on this metric.

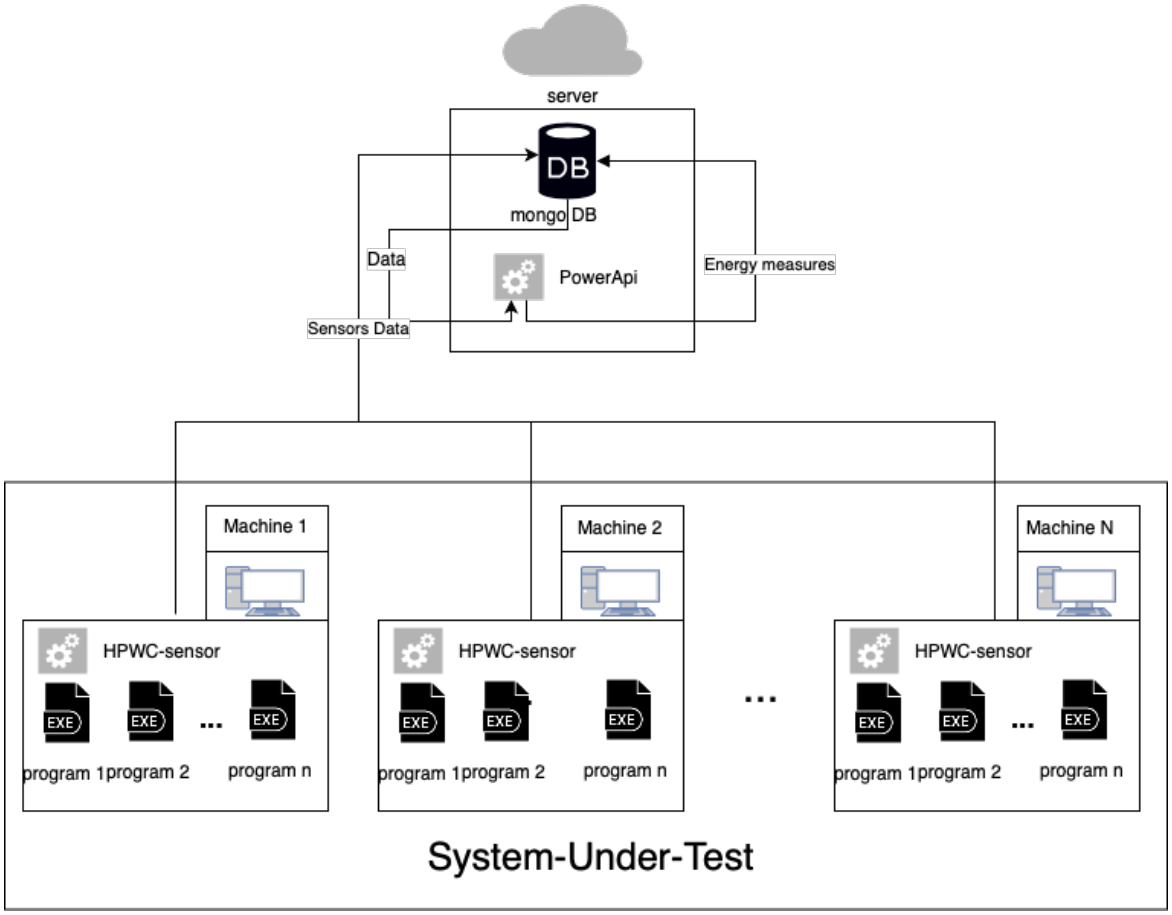


Figure 1.14: powerapi architecture

Energy measurement As we know, the energy of a program is the integrale of its power overtime. For us ower case, we used Intel *Running Power Average Limit* (RAPL) [9] to collect the power samples of the running tests,

We used used POWERAPI [5], to report Data collected by intel Rapl and send it into another machine that we call computing machine. then we calculate the Energy using the the trapezoidal rule.

1.15

$$E = \int_b^a P(t)dt \simeq \sum_{k=1}^{n} \frac{P(t_k - 1) + P(t_k)}{2} \tag{1.1}$$

Figure 1.15

figure 1.14 shows the architecture of our testing model.

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

1.6.3 tests preparation

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

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

- Isolation: each container has only the test program implemented with a single python runtime to remove any interference between different implementations.
- Deplpoyment: to use the testing machine without extra configurations that may alter the behaviour of the os toward the enrgy consumtion
- Reproductibility: One of the most frequent benchmark crimes [20] in research is the lack of Reproductibility, by using Docker we ensure that each test has an Image that will be accessible in public.

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

1.6.4 Extension

As we have done with the previous chapters. we provide a tool that allows to extend the tests with new workloads and new candidates. In the repository [28]. we have a tool that allows to generate new workloads and new candidates. The script `generator.py` allows to create new benchmarks by implementing a python code within different interpeters. then it generates

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

`launcher-benchmark.sh` that can be used directly to run the expermient. Furthermore all the successful implementations are stored in seperate directory and the that couldn't work (mostly because of compatibility issues) stored in a recap file called `benchmarkTest.md`. where benchmark is the name of the new workload; To add extra **candidates** one should add a base docker file that containes the new implementation and if there should be extra manipulation that should be added to the workload files ,such as adding new decorator or changing some parameters, then they should be added as an extra function in the script `generator.py`. Finally they should be included in the python candidates.

1.7 Results and finding

1.7.1 Preliminary stduies

This part will be dedicated to analyse the initial behaviour of different python interpereters. First we started with a typical benchmark binarrytree. where we compare the energy consumption of the different implementations.

1.8 placeholder

we performed a shapiro normality test for the first on the results, and almost all the p-values smaller than alpha=0.01% therefore the distrubution is not normal

for arrays and vectors graalpython took so much time that we decided to remove it

the startup cost for ironpython was too high for graalpython and ipy they couldnt handle big numbers hence the overflow when it comes to execution time but since we measure the energy outside the values of the energy aren't impacted

as usual there is correlaction between DRAM and CPU so no need to classify based on DRAM + the DRAM consumes less 10% of the energy compared to the CPU

so the plan is to prove that there is a statistical difference then we use the multiparameter optimization , howerver we stop in the phase where we calculate the score because we dont know the weight of each parameter (of tommti microbenchmark) and instead of doing the sum with the coefissients we use the radar plot to let the reader decide which one is the most suitable for his usage. howerver since the differences are gigantics we change the scale to logarithmic to be easier for the eye to read

As we can see in figure 1.18, there is no evolution between cpython2 and cpython3 Intelpython and active python both folows the same behaviour. one can conclude that the work that have been on those interpreters is mainly to improve a specific purpose, Active

benchmark	A	GG	LL	Or	XOR
activepython	676.980	763.208	651.783	743.016	728.828
cpython2	441.082	435.886	430.846	415.247	419.081
cpython3	595.209	685.085	563.839	657.972	655.560
cython	35.077	182.688	274.177	34.868	34.504
nuitka	33.260	32.980	33.256	33.472	33.030
numba2	9.102	8.411	9.460	9.375	9.755
numba3	9.566	10.144	9.219	9.344	9.665
pypy2	8.456	7.844	8.286	8.138	7.952
pypy3	7.552	8.093	8.108	8.669	8.623
shedskin	8.024	8.070	8.399	8.126	8.277

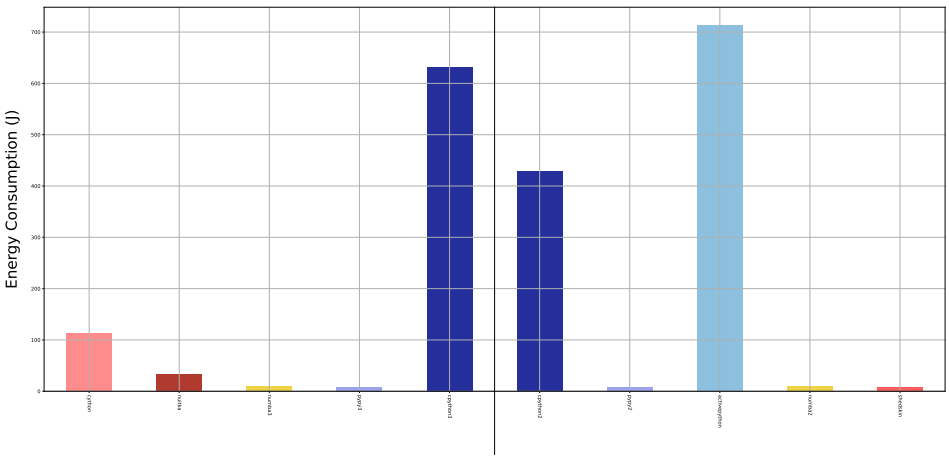


Figure 1.16: energy consumption of different impelementations using Bit Operation benchmarks (Joule)

Table 1.5: Energy consumption python environment

Implementation	array		intArithmetic		doubleArithmetic		hashes		heapsort		trig	
	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*
activepython	4e02	0.0079	6.9e02	0.0022	6.6e02	0.0043	9.7e02	0.0079	2.9e02	0.0079	5.1e02	0.0079
cpython2	3.4e02	0.0079	5.6e02	0.0022	5.5e02	0.0043	6.5e02	0.0079	2.9e02	0.0079	4.1e02	0.0079
cpython3	3.2e02	0.0079	7.4e02	0.0022	7.3e02	0.0043	7.9e02	0.0079	2.4e02	0.0079	4.1e02	0.0079
graalpython	1.2e05	0.0079	24	0.0022	24	0.0043	6.4e02	0.0079	1.3e05	0.0079	72	0.0079
intelpython2	3.7e02	0.0079	5.9e02	0.0022	5.6e02	0.0043	7.1e02	0.0079	2.7e02	0.0079	4.2e02	0.0079
intelpython3	3.5e02	0.0079	7.7e02	0.0022	7.6e02	0.0043	9.4e02	0.0079	2.6e02	0.0079	4.8e02	0.0079
ipy	3e02	0.0079	4.4e02	0.0022	4.7e02	0.0079	1.3e03	0.0079	2.6e02	0.0079	4.5e02	0.0079
jython	5.2e02	0.0079	1.3e02	0.0022	1.6e02	0.0043	6.4e02	0.0079	4.6e02	0.0079	6.2e02	0.0079
micropython	3.1e02	0.0079	8.2e02	0.0022	8.4e02	0.0079	9.5e03	0.0079	3.4e02	0.0079	5.3e02	0.0079
nuitka	2.9e02	0.0079	5.4e02	0.0022	5.5e02	0.0079	9.5e02	0.0079	2.2e02	0.0079	3.9e02	0.0079
numba2	1.9e02	0.0079	27	0.0022	36	0.0079	6.8e02	0.0079	2.1e03	0.0079	4.4e02	0.0079
numba3	**11**	**best**	10	0.0022	10	0.0079	9.1e02	0.0079	7.2e02	0.0079	4.5e02	0.0079
pypy2	16	*0.15*	29	0.0022	30	0.0043	**1.1e02**	**best**	**20**	**best**	64	0.0079
pypy3	13	*0.15*	17	0.0022	18	0.0079	1.9e02	0.0079	**20**	0.31	65	0.0079
shedskin	47	0.0079	**7.4**	**best**	**7.3**	**best**	1.1e03	0.0079	44	0.0079	**7.7**	**best**

Implementation	longArithmetic		matrixMultiply		io		stringConcat		nestedLoop		except	
	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*	energy (J)	*p-values*
activepython	6.7e02	0.0079	4.3e02	0.0079	2.1e02	0.0079	14	0.0079	4.1e02	0.0079	2.6e02	0.0079
cpython2	5.5e02	0.0079	4e02	0.0079	2e02	0.0079	12	0.0079	4.2e02	0.0079	4.3e02	0.0079
cpython3	7.4e02	0.0079	4.4e02	0.0079	2e02	0.0079	13	0.0079	3.8e02	0.0079	2.3e02	0.0079
graalpython	24	0.0079	45	0.0079	7.5e02	0.0079	26	0.0079	12	0.0079	1.6e02	0.0079
intelpython2	5.7e02	0.0079	4.8e02	0.0079	2e02	0.0079	13	0.0079	4.4e02	0.0079	4.6e02	0.0079
intelpython3	7.7e02	0.0079	5e02	0.0079	2.2e02	0.0079	14	0.0079	4.4e02	0.0079	2.7e02	0.0079
ipy	4.7e02	0.0079	4.1e02	0.0079	3.8e02	0.0079	49	0.0079	3.3e02	0.0079	7.2e02	0.0079
jython	1.6e02	0.0079	1.9e02	0.0079	2e02	0.0079	21	0.0079	1.9e02	0.0079	7.1e02	0.0079
micropython	8.3e02	0.0079	5.3e02	0.0079	5.3e03	0.0079	44	0.0079	4.3e02	0.0079	3.6e02	0.0079
nuitka	5.4e02	0.0079	4.3e02	0.0079	2e02	0.0079	12	0.0079	3.6e02	0.0079	2.2e02	0.0079
numba2	35	0.0079	4e02	0.0079	2.2e02	0.0079	20	0.0079	14	0.0079	4.5e02	0.0079
numba3	11	0.0079	4.2e02	0.0079	2.1e02	0.0079	19	0.0079	10	0.0079	2.4e02	0.0079
pypy2	30	0.0079	24	0.056	**1.7e02**	**best**	7.7	**best**	27	0.0079	**14**	**best**
pypy3	17	0.0079	**23**	**best**	2.6e02	0.0079	8.3	0.69	23	0.0079	14	0.69
shedskin	**7.8**	**best**			3.8e02	0.0079	44	0.0079	7.1	**best**	5.6e02	0.0079

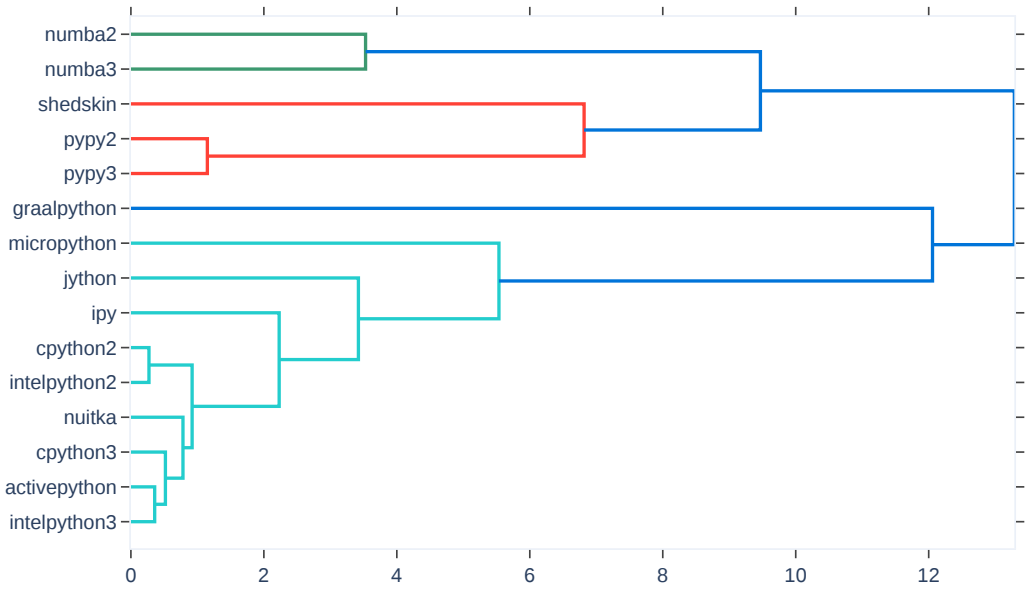


Figure 1.17: Dendogram of the different implementations


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

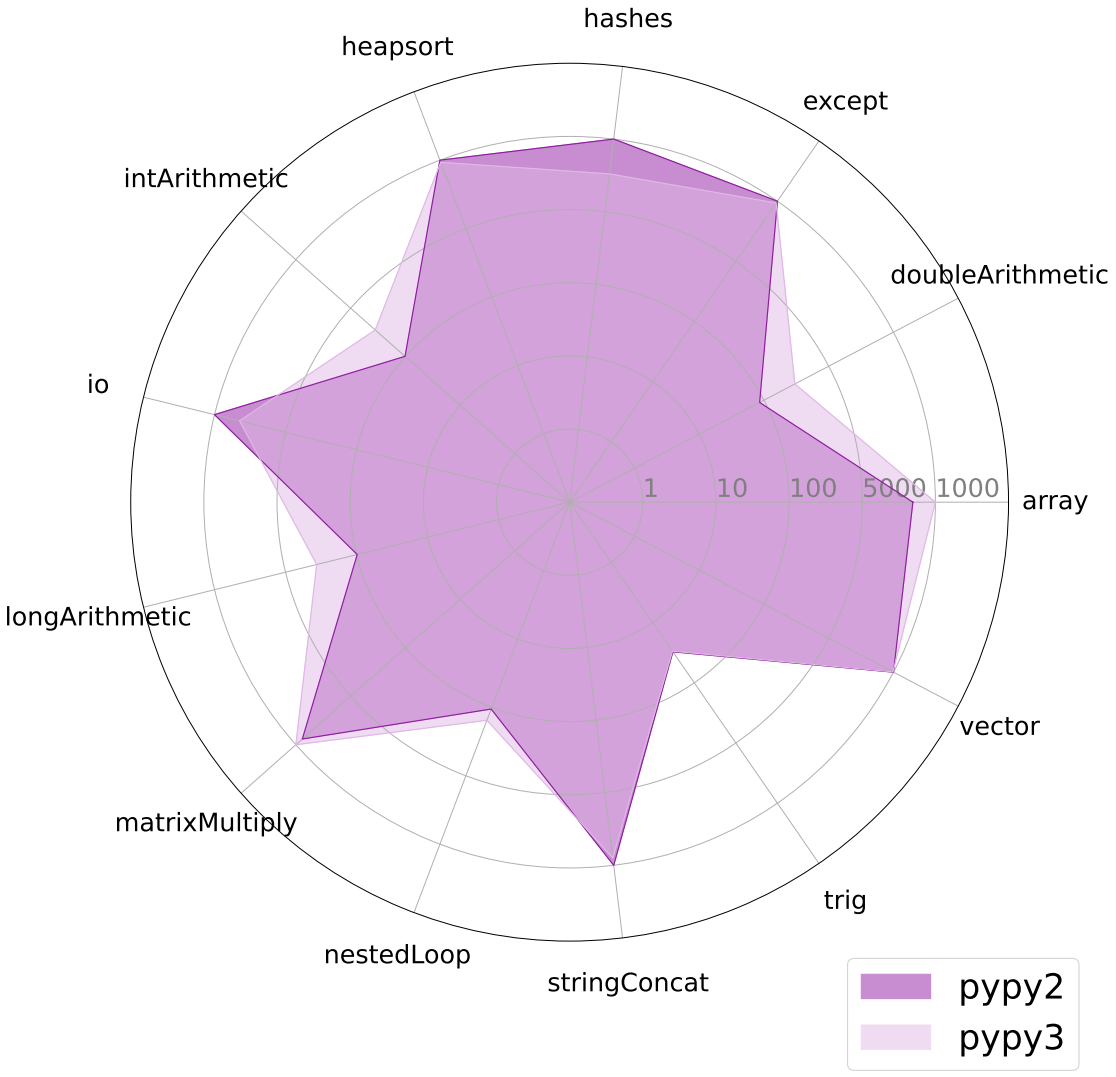


Figure 1.19: green factor of pypy

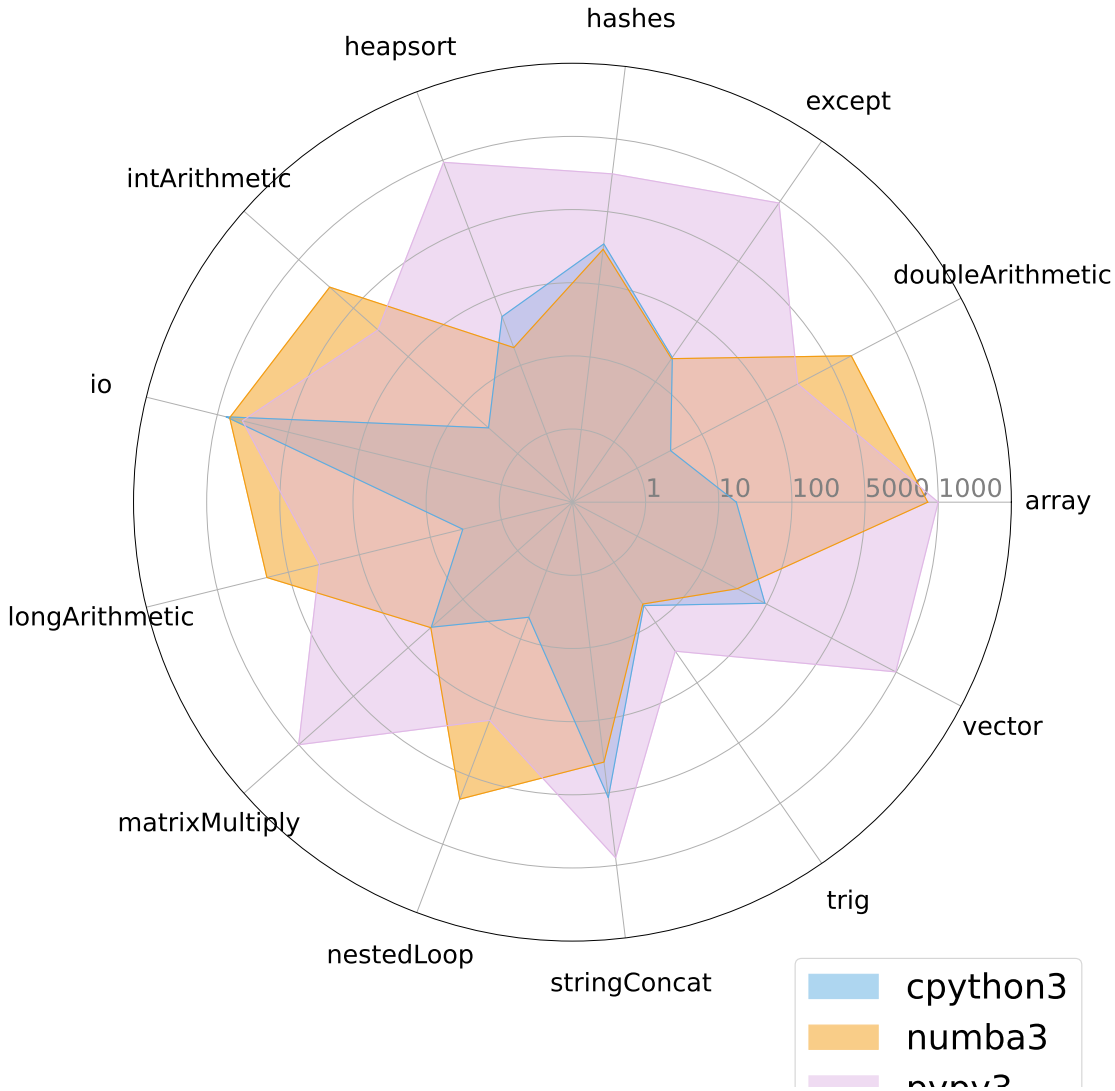


Figure 1.20: comparaison of pypy vs python vs numba

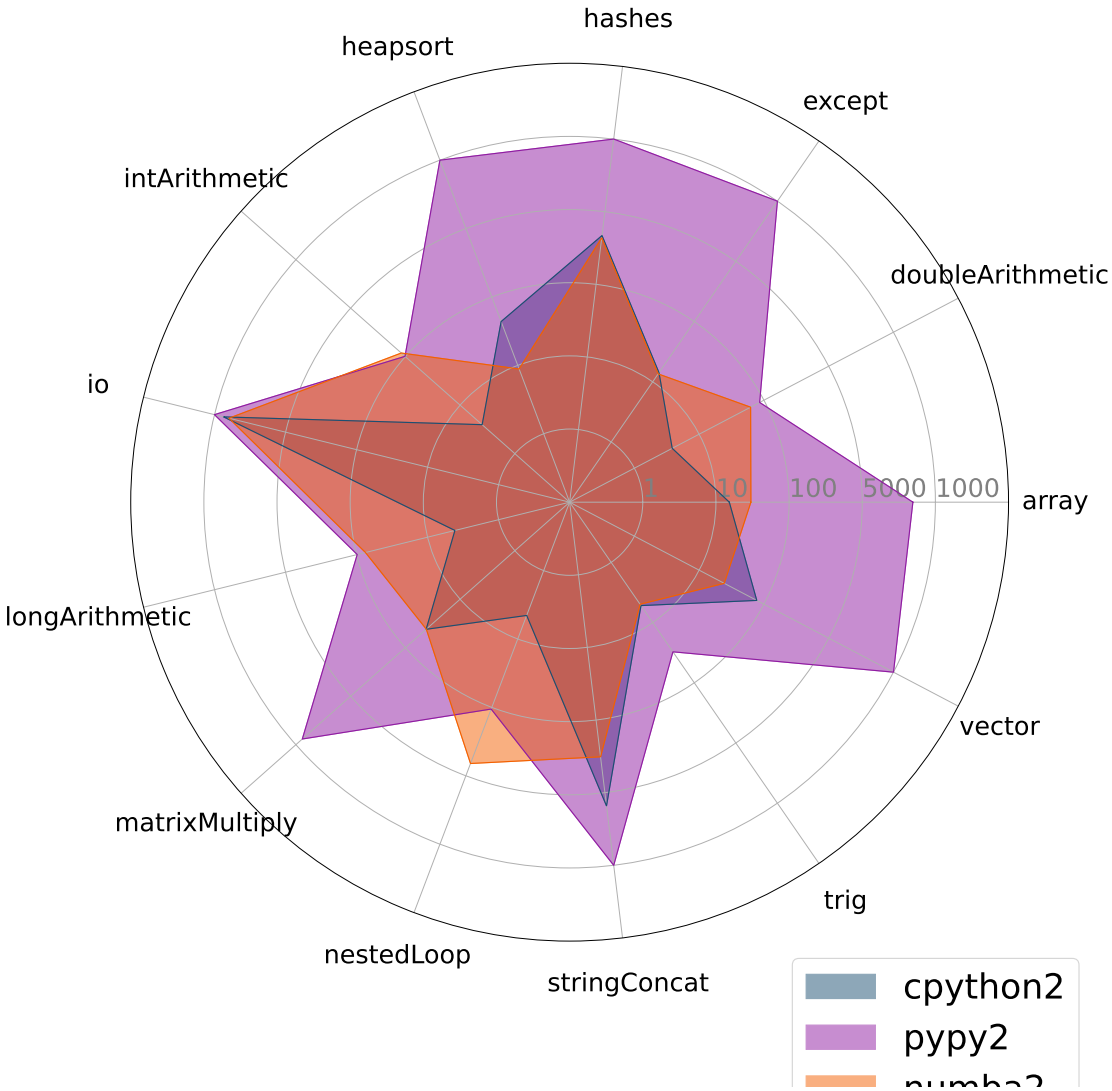


Figure 1.21: comparaison of pypy vs python vs numba

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

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

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

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

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

1.9 conclusion

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

Chapter 2

The Impact of Java Virtual Machine on Energy Consumption

2.1 Introduction

As reported in the state of the art, Java is one of the most popular programming languages adopted by practitioners.Furthermore, if we take into consideration legacy applications, Java becomes the most used programming language.In addition to its popularity, Java exhibits an interesting behavior when it comes to energy consumption and performance, Java applications can be at the same time one of the most energy-efficient or hungry solutions. As we have seen in the previous chapter, an inappropriate combination of parameters can drive Java applications from the top language to the bottom just by setting the wrong parameters. Therefore, we wanted to dig deeper into this aspect of Java and study its runtime. This chapter thus focuses on the impact of the runtime of Java applications on energy consumption.

2.1.1 Goal

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

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

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

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

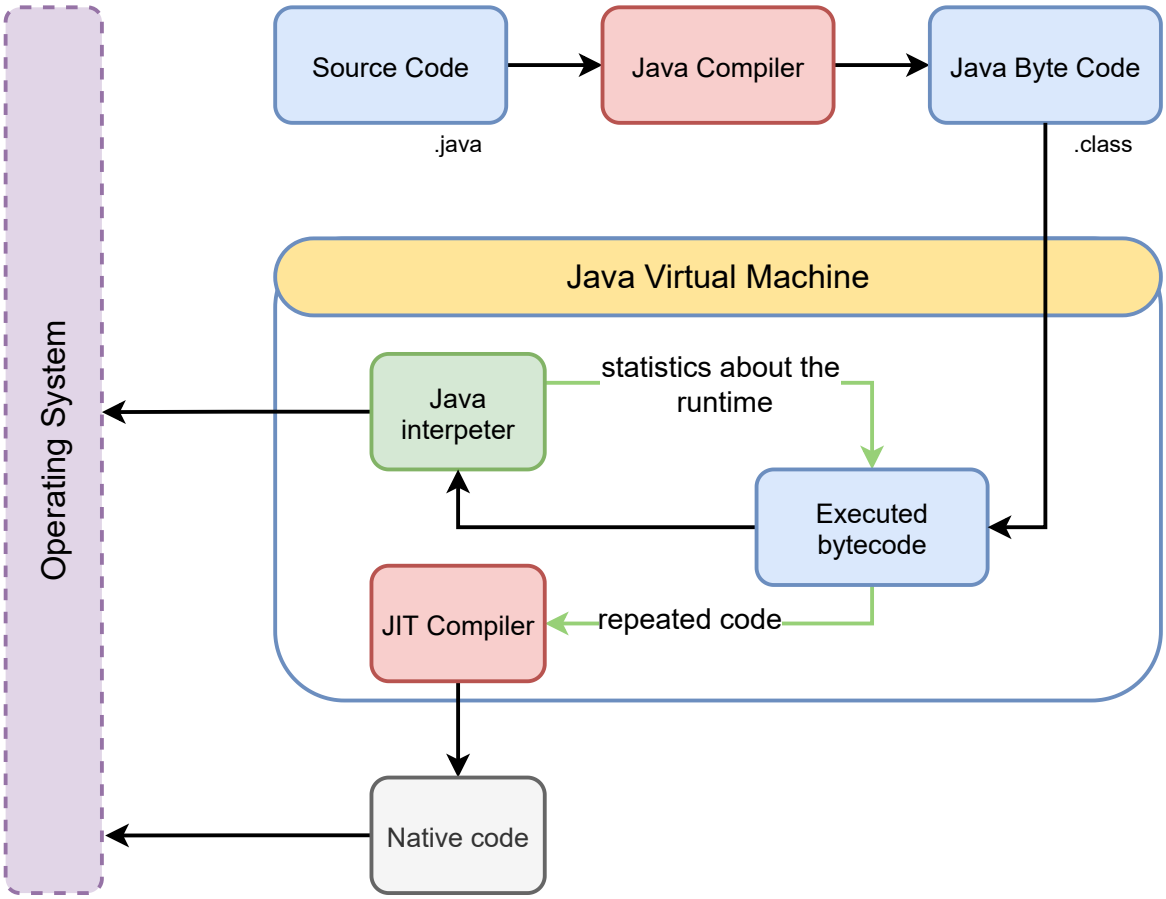


Figure 2.1: JVM architecture

2.1.2 definition of the JVM

2.2 Experimental Protocol

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

2.2.1 Measurement Contexts

Software Settings. For the sake of reproducibility, each experiment runs within a Docker container based on SDKMAN![1] image and Alpine docker.[2]

[1]https://sdkman.io
[2]https://github.com/alpinelinux/docker-alpine

Table 2.1: 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

Hardware Settings. To report on reproducible measurements, we used the cluster Dahu from the G5K platform [3] 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 [15].

Java Virtual Machines Candidates. We considered a set of 52 JVM distributions taken from 8 different providers/packagers mostly obtained from SDKMAN!, as listed in Table 2.1. Depending on providers, either all the versions, majors, or LTS are made available by SDKMAN!.

2.2.2 Workload

We ran our experiments across 12 Java benchmarks we picked from OpenBenchmarking.org.[3] This includes 5 acknowledged benchmarks from the DACAPO benchmark suite v. 9.12 [4], 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 [10, 8]. 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 [18, 19], 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

[3] https://openbenchmarking.org

Table 2.2: List of selected open-source Java benchmarks taken from DACAPO and RENAISSANCE.

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

of programming paradigms, including concurrent, parallel, functional, and object-oriented programming. Table 2.2 summarizes the selected benchmarks with a short description.

Figure 2.2 highlights the scope of each benchmark from the test suite.

2.2.3 Metrics and Measurement

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

- execution time,
- number of threads.

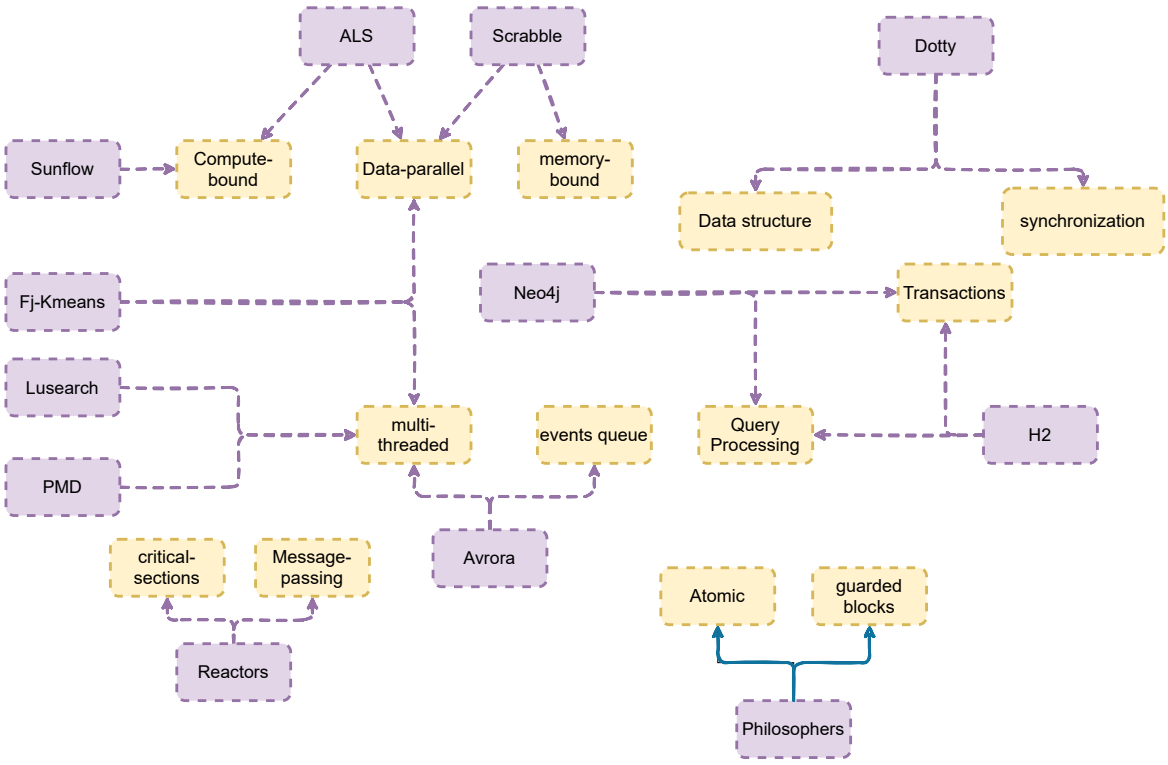


Figure 2.2: Target scope of DACAPO and RENAISSANCE benchmarks.

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 [9, 6]. We note that, due to CPU energy consumption variations issues [15], we used the same node for all our experiments. Moreover, we tried to be very careful, while running our experiments, not to fall in the most common benchmarking "crimes" [21]. Every single experiment, therefore, reports on energy metrics obtained from at least 20 executions of 50 iterations per benchmark. All of our experiments are available for use/reproducibility from our anonymous repository.[4]

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

2.2.4 Extension

We also added an extension to the protocol to allow the user to run the same experiment with different configurations. The package is available in the GitHub repository.[5] To add extra **jvm candidates** for the benchmark applications, we added a new configuration file

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

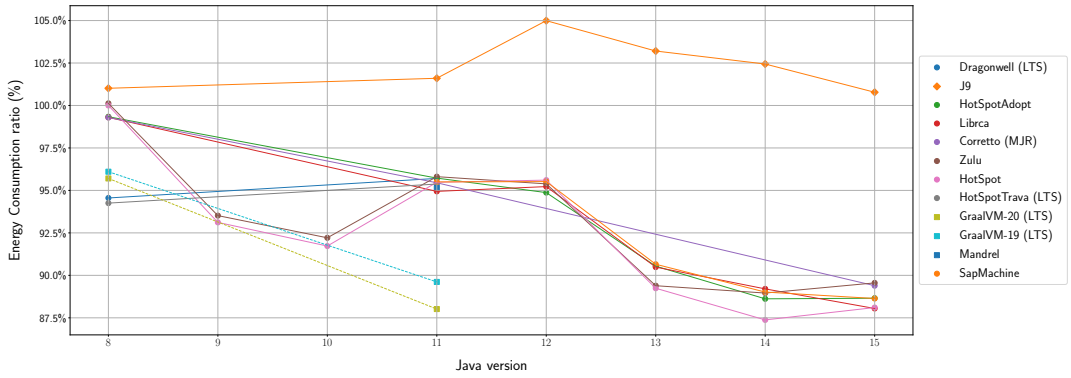


Figure 2.3: Energy consumption evolution of selected JVM distributions along versions.

`jvm.sh` in the root directory of the repository, where we put the name and the version of the jvm to be used.***TODO : The JVM must be supported by SDKMAN!*** For the **input workload**,the benchmarks should be provided in the `benchmarks` directory. As for extra **metrics**, one can create a new script file that monitors the experiment and record the metrics. We provide some examples, such as `recordpower.sh` to measure the instantaneous power and `recordthreads.sh` to measure the number of active threads during the experiment.

For faster experiments, we propose JREFERRAL[6], an open source tool that automatically compares the JVM energy consumption and recommends the most energy-efficient configuration for a given Java application. We further discuss this tool in Section **??**.

2.3 Experiments & Results

2.3.1 Energy Impact of JVM Distributions

Job-oriented applications. To answer our first research question, we executed 62,400 experiments by combining the 52 JVM distributions with the 12 Java benchmarks, thus reasoning on 100 energy samples acquired for each of these combinations. Figure 2.3 first depicts the accumulated energy consumption of the 12 Java benchmarks per JVM distribution and major versions (or LTS when unavailable). Concretely, We measure the energy consumption of each of the benchmarks and compute the ratio of energy consumption compared to HOTSPOT-8, which we consider as the baseline in this experiment. Then, we sum the ratios of the 12 benchmarks and depict them as percentages in Figure 2.3.

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

One can observe that, along with time and versions, the energy efficiency of JVM distributions tends to improve (10% savings), thus demonstrating the benefits of optimizations delivered by the communities. Yet, one can also observe that energy consumption may differ from one distribution to another, thus showing that the choice of a JVM distribution may have a substantial impact on the energy consumption of the deployed software services. For example, one can note that J9 can exhibit up to 15% of energy consumption overhead, while other distributions seem to converge towards a lower energy footprint for the latest version of Java. As GRAALVM adopts a different strategy focused on LTS support, one can observe 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 2.3: J9, the HOTSPOT and its variants, and GRAALVM. Additional detailed figures to illustrate the evolution of energy consumption per benchmark/JVM are made available from the online repository.[7]

Then, Figure 2.4 depicts the evolution of the energy consumption of the 12 benchmarks, when executed on the HOTSPOT JVM. Figure 2.4 reports on the energy consumption variation of individual benchmarks, using to HOTSPOT-8 as the baseline. Our results show that the JVM version can severely impact the energy consumption of the application. However, unlike Figure 2.3, one can observe that, depending on applications, the latest JVM versions can consume less energy (60% less energy for Scrabble) or more energy (25% more energy for the Neo4J). It is worth noticing that the energy consumption of some benchmarks, such as Reactors, exhibit large variations across JVM versions due to experimental features and changes that are not always kept when releasing LTS versions (version 11 here). For example, the introduction of `VarHandle` to allow low-level access to the memory order modes available in JDK 9 and work along `Unsafe Class` was removed from JVM 11.[8]

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

[7] https://anonymous.4open.science/r/jvm-comparaison-213E/Readme.md
[8] https://blogs.oracle.com/javamagazine/the-unsafe-class-unsafe-at-any-speed

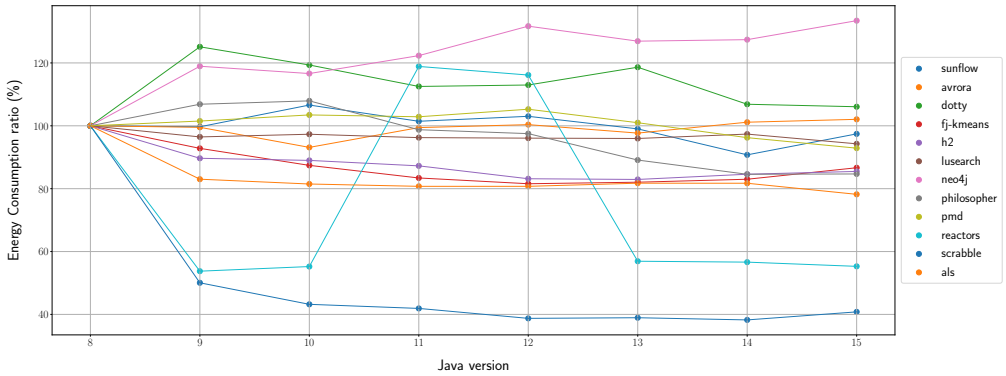


Figure 2.4: Energy consumption of the HotSpot JVM along versions.

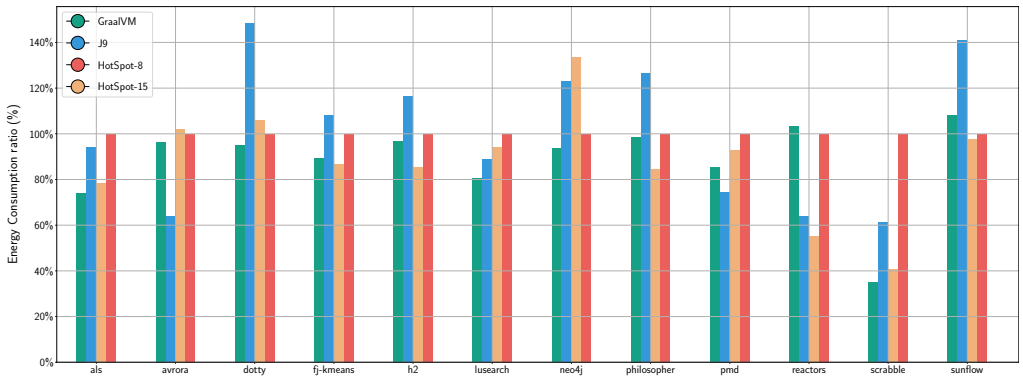


Figure 2.5: Energy consumption comparison across Java benchmarks for HOTSPOT, GRAALVM & J9.

Figure 2.5 further explores the comparison of energy efficiency of the JVM distributions per benchmark. One can observe that, depending on the benchmark's focus, the energy efficiency of JVM distributions may strongly vary. When considering individual benchmarks, J9 performs the worst for at least 6 out of 12 benchmarks—*i.e.*, the worst ratio among the 4 tested distributions. Even though, J9 can still exhibit a significant energy saving for some benchmarks, such as Avrora, where it consumes 38% less energy than HOTSPOT and others.

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

Table 2.3: Power per request for HOTSPOT, GRAALVM & J9.

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

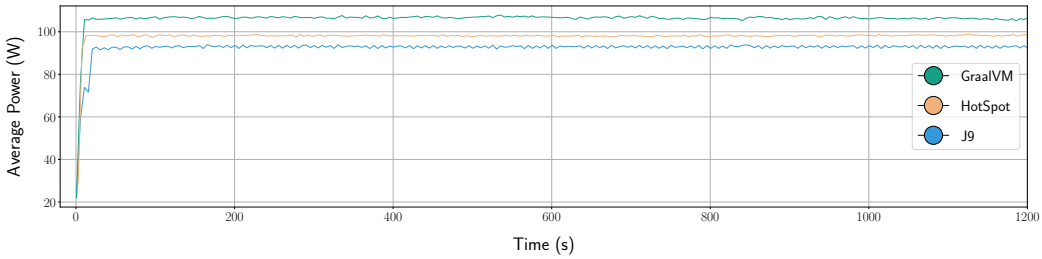


Figure 2.6: Power consumption of Scrabble as a service for HOTSPOT, GRAALVM & J9.

Service-oriented applications. In this section, instead of considering bounded execution of benchmarks, we run the same benchmarks as services for 20 minutes, and we compare the average power and total requests processed by each of the 3 JVM distributions. Globally, the results showed that the average power when using GRAALVM, HOTSPOT, and OPENJ9 is often equivalent and stable over time. This means that the energy efficiency observed for some JVM distributions with Job-oriented applications is mainly related to shorter execution times, which incidentally results in energy savings. Nonetheless, we can highlight two interesting observations for two benchmarks whose behaviors differ from others. First, the analysis of the Scrabble benchmark experiments showed that, in some scenarios, some JVMs can exhibit different power consumptions. Figure 2.6 depicts the power consumed by the 3 JVM distributions for the Scrabble benchmark. One can clearly see that GRAALVM requires an average power of 109 W, which is 9 W higher than HOTSPOT-15 and 15 W higher than J9. When it comes to the number of requests processed by Scrabbles during that same amount of time, GRAALVM completes 5,336 requests, against 3,595 for HOTSPOT and 2,603 for J9, as shown in Table 2.3. 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 2.5 for this benchmark.

The second interesting situation was observed on the Dotty benchmark. More specifically, during the first 100 seconds of the execution of the Dotty benchmark on all evaluated JVMs.

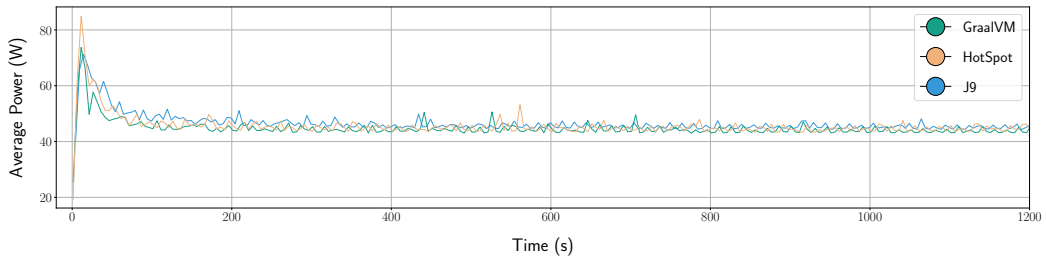


Figure 2.7: Power consumption of Dotty as a service for HOTSPOT, GRAALVM & J9.

At the beginning of the execution, GRAALVM has a slightly lower power consumption, is faster, and consumes 10% less energy. After about 150 seconds, the power differences between the 3 JVMs is barely noticeable. One can, however, notice the effect of the JIT, as HOTSPOT takes the advantage over GRAALVM and becomes more energy efficient. In total, HOTSPOT completes 597 requests against 510 for GRAALVM and 381 for J9, as shown in Table 2.3. HOTSPOT was thus the best choice in the long term, which explains why it is always necessary to consider a warm-up phase and wait for the JIT to be triggered before evaluating the effect of the JVM or the performance of an application. This is exactly what we did in our experiments, and why HOTSPOT was more energy efficient than GRAALVM in Figure 2.5, thus ignoring the warm-up phase would have been misleading.

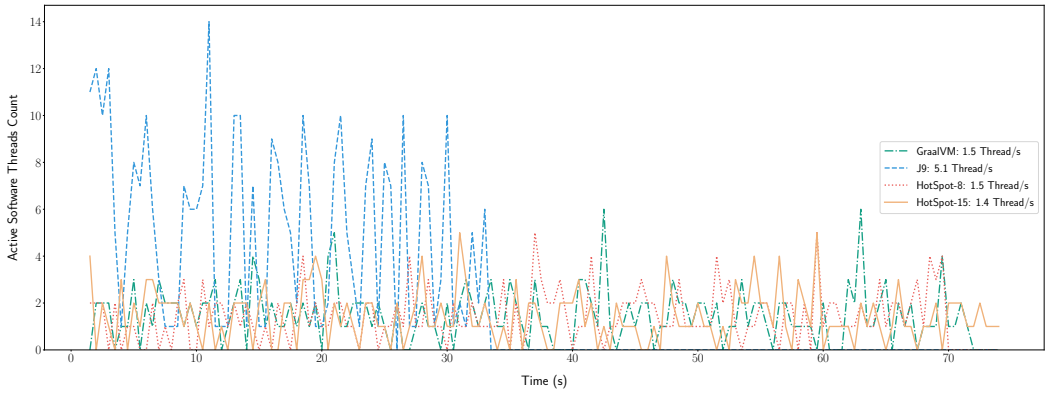
To answer **RQ 1**, we conclude that—while most of the JVM platforms perform similarly—we can cluster JVMs in 3 classes: HOTSPOT, J9, and GRAALVM. The choice of one JVM of these classes can have a major impact on software energy consumption, which strongly depends on the application context. When it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

2.3.2 Energy Impact of JVM Settings

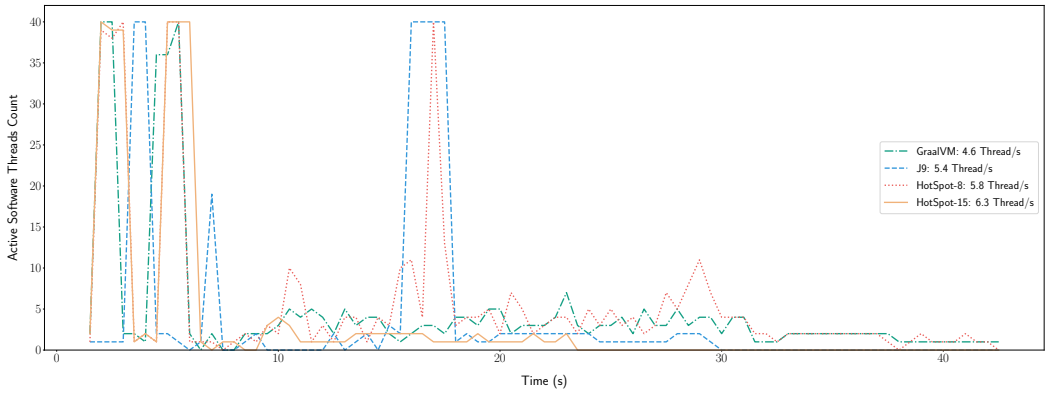
The purpose of our study is not only to investigate the impact of the JVM platform on energy consumption, but also the different JVM parameters and configurations that might have a positive or negative effect, with a focus on 3 available settings: multi-threading, JIT, and GC.

Multithreading

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



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



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

Figure 2.8: Active threads evolution when using HOTSPOT, GRAALVM, or J9.

Investigating such a hypothesis requires a selection of highly parallel and CPU-intensive benchmarks, which is one of the main criteria for our benchmark selection. As no tool can accurately monitor the energy consumption at a thread level, we monitor the global power consumption and CPU utilization during the execution using RAPL for the energy, and several Linux tools for the CPU-utilization (`htop`, `cpufreq`). Knowing that most of the benchmarks are multi-threaded jobs that use multiple cores, further analysis of thread management is required to understand the results of our previous experiments. We thus selected the benchmarks that highlighted the highest differences along JVM distributions from Figure 2.5, namely Avrora and Reactors. We studied their multi-threaded behavior to optimize their energy efficiency.

Figure 2.8 delivers a closer look to the thread allocation strategies adopted by JVM. First, Figure 2.8a illustrates the active threads count evolution over time (excluding the JVM-related threads, usually 1 or 2 extra threads depending on the execution phase) for Avrora. One can notice through the figure that J9 exploits the CPU more intensively by running much more

parallel threads compared to other JVMs (an average of 5.1 threads per second for J9 while the other JVMs do not exceed 1.5 thread per second). Furthermore, the number of context switches is twice bigger for J9, while the number of soft page faults is twice smaller. The efficient J9 thread management explains why running the Avrora benchmark took much less time and consumed less energy, given that no other difference for the JIT or GC configuration was spotted between the JVMs. Another key reason for the J9's efficiency for the Avrora benchmark is memory allocation, as OpenJ9 adopts a different policy for the heap allocation. It creates a non-collectible *thread local heap* (TLH) within the main heap for each active thread. The benefit of cloning a dedicated TLH is the fast memory access for independent threads: each thread has its heap and no deadlock can occur.

The second example in Figure 2.8b depicts the active threads evolution over time of the Reactors benchmark. In this case, all the JVMs have a close average of threads per second. Nevertheless, one can still observe that HOTSPOT-15 and J9 keep running faster, which confirms the results of Figure 2.5, where both JVMs consume much less energy compared to GRAALVM and HOTSPOT-8. This difference in energy consumption between benchmarks can be less likely caused by thread management for the Reactors benchmark, as HOTSPOT-8 reports on a higher average of active threads. However, the TLH mechanism was not as efficient as for the Avrora benchmark, as dedicating a heap for each thread can also cause some extra memory usage for data duplication and synchronization, especially if a lot of data is shared between threads.

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

Just-in-Time Compilation

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

For J9, we considered fixing the intensity of the JIT compiler at multiple levels (cold, warm, hot, veryhot, and scorching).[9] The hotter the JIT, the more code optimization to be triggered. We also varied the minimum count method calls before a JIT compilation occurs (10, 50, 100), and the number of JIT instances threads (from 1 to 7). For HOTSPOT-15, we conducted experiments while disabling the tiered complication (that generates compiled versions of methods that collect profiling information about themselves), and we also varied the JIT maximum compilation level from 0 to 4, we also tried out HOTSPOT with a basic

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

GRAALVM JIT. We note that level 0 of JIT compilation only uses the interpreter, with no real JIT compilation. Levels 1, 2, and 3 use the C1 compiler (called client-side) with different amounts of extra tuning. The JIT C2 (also called server-side JIT) compiler only kicks in at level 4.

For GRAALVM, we conducted experiments with and without the JVMCI (a Java-based JVM compiler interface enabling a compiler written in Java to be used by the JVM as a dynamic compiler). We also considered both the community and economy configurations (no enterprise). A JIT+AOT (*Ahead Of Time*) disabling experiment has also been considered for all of the 3 JVM platforms. Table 2.4 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, where the values in green highlight the strategies that consumed significantly less energy than the default (less energy and significant *p*-value).

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

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

For HOTSPOT, keeping the default configuration of the JIT is also mostly good. The usage of the C2 JIT is often beneficial (JIT level 4) in most cases while using the GRAALVM JIT reported similar energy efficiency. Yet, some benchmarks showed that using only the C1 JIT (JIT level 1) is more efficient and even outperforms the usage of the C2 compiler. 10% on Avrora and 30% on Pmd are examples of energy savings observed by using the C1 compiler. However, being limited to the C1 compiler can also cause a huge degradation in energy consumption, such as 32% and 34% of additional energy consumed for the Dotty and FJ-kmeans benchmarks, respectively. Hence, if it is a matter of not using the C2 JIT, the experiments have shown that the level 1 JIT is always the best, compared to levels 2 or 3 that

Table 2.4: Energy consumption when tuning JIT settings on HOTSPOT, GRAALVM & J9

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

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

Table 2.5: 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

also use the C1 JIT, but with more options, such as code profiling that impacts negatively the performance and the energy efficiency. Level 0 JIT compilation should never be an option to consider. No *p*-value has been computed for Level 0, due to the limited amount of iterations executed with this mode (very high execution time, clearly much more consumed energy).

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

Garbage Collection

Changing or tuning the GC strategy has been acknowledged to impact the JVM performances [11]. To investigate if this impact also benefits energy consumption, we conducted a set of experiments on the selected JVMs. We considered different garbage collector strategies with a limited memory quantity of 2 GB, and recorded the execution time and the energy consumption. The tested GC strategies options mainly vary between J9 and the other 2 JVMs, as detailed in Table 2.5.

For HOTSPOT and GRAALVM, we also considered many GC policies, as described in Table 2.6. 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 2.7 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

Table 2.6: 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

consumption being equal to the default configuration with a 95% confidence. The p-values in bold show the values that are significantly different from the default configuration, whereas the values in green highlight the strategies that consumed significantly less energy than the default. For GRAALVM, one can see that the GC default configuration is efficient in most experiments, compared to other strategies. The main noticeable impact is related to the ParallelGC and ParallelOldGC. In fact, the ParallelGC can be 13% more energy efficient in some applications with a significant p-value, such as Reactors, compared to default. However, the same GC strategy can cause the software to consume twice times more, as for the Neo4j benchmark, due to the high communications between the GC threads, and the fragmentation of the memory.

For J9, the default Gencon GC causes the software to report an overall good energy efficiency among the tested benchmarks. However, other GC can cause better or worse energy consumption than Gencon depending on workloads. Using the Metronome GC consumes 35% less energy for the ALS benchmark and 17% less energy for the Sunflow benchmark, but it also consumes 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

Table 2.7: Energy consumption when tuning GC settings on HOTSPOT, GRAALVM & J9

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

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

about three times the energy consumption, compared to the default Gencon GC. On the other hand, the Optthruput GC, which stops the application longer and less frequently, gave very good overall results and sometimes even outperformed the Gencon GC by a small margin. Other JVM parameters, such as the ConcurrentScavenge or noAdaptiveTenure did not have a substantial impact during our experiments.

Finally, the results of HOTSPOT shared similarities with GRAALVM. The ParallelGC happened to give better (6% for Dotty) or worst (10% for Neo4j) energy efficiency compared to the default GC. On the other hand, ParallelOldGC and Serial GC gave better results than the default G1 GC. More specifically, the second one consumed 30% and 6% less energy than the default GC for the Neo4j and Dotty benchmarks, respectively. The most interesting result for HOTSPOT is the 30% energy reduction obtained with the Serial GC. This last was also more efficient on ALS (6% less energy), compared to the default G1 GC, due to its single-threaded GC that only uses one CPU core.

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

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

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

2.4 Threats to Validity

Several issues may affect the validity of our work. First, we have the use of the Intel RAPL, one of the most accurate available tools to measure the energy consumption of software [9, 6]. However, RAPL only gives the global energy consumption and no fine-grained measures at process or thread levels. We used bare-metal hardware with a minimal OS and turned off all the non-essential services and daemons to limit the overhead that the OS may add to the execution, even if it is not substantial [15].

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

Benchmarks' execution time could also constitute a more subtle threat to the validity of our work, especially for some benchmarks that run fast, such as the Pmd benchmark. We thus gave a lot of attention to how long the benchmark is running for the hardware we used, and we tuned the input data workloads to execute benchmarks for at least many (from 10 to hundreds) seconds. Experiments ran at least 30 times to compute the average consumption and the associated standard deviation, therefore reasoning over reasonable dispersion around the average.

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

2.5 Tools and contributions

Table 2.8 illustrates an example of the final report of J-Referral. The tool was tested for 2 Java projects: Zip4J[10] and K-nucleotide.[11] Zip4J runs a large file compression, while K-nucleotide extracts a DNA sequence, and updates a hashtable of k-nucleotide keys to count specific values. The short report presented in Table 2.8 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

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

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

Table 2.8: 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

vs. CPU consumption, etc. The tool is available as *open-source software* (OSS) from our GitHub repository.[12]

2.6 Conclusion

The results of our investigations showed that many JVMs share energy efficiencies and can be 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 observe a unique champion when it comes to energy consumption, GRAALVM reported the best energy efficiency for a majority of benchmarks. Nonetheless, each JVM can achieve the best or the worst depending on the hosted application. One cause can be thread management strategies, as observed with J9 when advantageously running Avrora. Moreover, some JVM settings can cause energy consumption variations. Our experiments showed that the default JIT compiler of the JVM is often near-optimal, in at least 80% of our experiments. The default GC, however, was outperforming alternative strategies in half of our experiments, with some large gains observed when using some alternative GC depending on the application characteristics.

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

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

Bibliography

[noa] PYPL PopularitY of Programming Language index. https://pypl.github.io/PYPL.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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