Eco-conception des logiciels: Investiguer et réduire la consommation énergétique des logiciels

THÈSE DE DOCTORAT

Pour obtenir le titre de

Docteur en sciences

Domaine : INFORMATIQUE

Défendue le lundi 8 novembre 2021

Zakaria OURNANI

Université de Lille / SPI-MADIS

Software Eco-Design: Investigating and Reducing the Energy Consumption of Software

PHD THESIS

To obtain the title of

PhD of Science

Field : COMPUTER SCIENCE

Defended on Monday, November, 8th 2021

Zakaria OURNANI

University of Lille / SPI-MADIS

Thesis committee:

Thesis Directors:	Romain ROUVOY	-	University of Lille
	Joël PENHOAT	-	Orange
Supervisor:	Pierre RUST	-	Orange
Reviewers:	Anne-Cécile ORGERIE	-	IRISA Rennes
	Thomas LEDOUX	-	IMT-Atlantique Nantes
Examiner:	Sophie QUINTON	-	INRIA Grenoble
Chair:	Anne ÉTIEN	-	University of Lille

Acknowledgements

First of all, I dedicate this work to my new born daughter, with plenty of love. To my beloved wife who supported me a lot to acheive my thesis. To my one and unique mom that made me who i am. To my dad that was always very supportive. And to my younger brother.

Then, I would like to thank the following people, without whom I would not have been able to complete this research and obtain my degree! I would like to thank my supervisors Romain ROUVOY, Pierre RUST and Joel PENHOAT for the huge support they provided me during my thesis, and being there to encourage me and guide me through this adventure. I would also like to thank persons i collaborated with during my thesis, including Arnaud DIQUELOU, Chakib BELGAID, Guillaume FIENI, Jean Remy FALLERI and every other person participating in my research and interviews.

Finally, i would like to thank the reporters and reviewers, for accepting to be part of the jury, and spare time to read and review my work.

Abstract

Energy consumption is an emerging concern in multiple domains and fields, including ICT and data centers. In fact, the energy consumption of data centers has drastically increased in the last decade for both hardware and software entities, especially due to the democratization of cloud services and the huge amount of transiting data. Formerly, The energy consumption was mainly related to the used hardware, and its capacity to maintain a low power consumption while achieving tasks. However, the running software is in fact as important as hardware, and is as responsible for very substantial gains or drawbacks in energy consumption.

The ultimate goal of this thesis is to help developers and practitioners understand and actively think about green software design in their work, in order to reduce the energy consumption of their software and deliver energy efficient products. We thus contribute to supplement green software design knowledge.. To achieve this, we start with conducting a qualitative study with developers, to discuss the multiple hurdles they are facing and their requirements to promote green software design within companies.

To reduce software energy consumption, practitioners have to measure it and track its evolution first. In our second contribution we investigate the problem of energy consumption variations. We provide guidelines on controllable factors that one could easily tune to reduce this variation and conduct steady and reproducible energy measurements.

Once practitioners are able to measure the energy consumption of their software, they can work on reducing it and produce energy efficient software. Thus, this thesis delivers 3 more contributions, focusing on the Java language. The first contribution aims at helping developers choose and configure their execution environment. We identified substantial differences in energy consumption using multiple JVM platforms with different JIT and GC configurations for different use cases. The second and third contributions study the impact on energy consumption of small changes that developers often apply on their source code (code refactoring and API/methods substitutions respectively). We show through these studies that structure oriented code refactorings do not substantially alter software energy consumption. On the other hand, Java I/O methods substitution drastically changed the energy consumption depending on the use case.

This thesis contributes to enrich the knowledge on green software design and provides insights and approaches to enhance the energy efficiency at multiple levels of software development.

Résumé

La consommation d'énergie est une préoccupation émergente dans de nombreux domaines, y compris pour les technologies d'informations et communications et les centres de données. En effet, la consommation d'énergie des centres de données a considérablement augmenté au cours de la dernière décennie, tant pour les entités matérielles que logicielles, notamment en raison de la démocratisation des services en ligne et de l'énorme quantité de données qui transitent. Auparavant, la consommation d'énergie était principalement liée au matériel utilisé, et à sa capacité à maintenir une faible consommation d'énergie pour réaliser des tâches. En réalité, le logiciel est aussi important que le matériel, et il est autant responsable d'une diminution ou augmentation de l'énergie consommée.

L'objectif ultime de cette thèse est d'aider les développeurs et les praticiens à comprendre et à introduire la conception de logiciels verts dans leur travail, afin de réduire la consommation d'énergie de leurs logiciels et de fournir des produits économes en énergie. Nous contribuons ainsi à compléter les connaissances sur la conception de logiciels verts. Pour y parvenir, nous commençons par mener une étude qualitative auprès des développeurs, afin de discuter des multiples obstacles auxquels ils sont confrontés et de leurs besoins pour promouvoir la conception de logiciels verts au sein des entreprises.

Pour réduire la consommation énergétique des logiciels, les praticiens doivent d'abord être capable de la mesurer et suivre son évolution. Dans notre deuxième contribution, nous étudions le problème des variations de la consommation d'énergie. Nous fournissons des lignes directrices sur des facteurs contrôlables que l'on peut facilement actionner pour réduire cette variation et effectuer des mesures énergétiques stables et reproductibles.

Une fois que les praticiens sont capables de mesurer la consommation d'énergie de leurs logiciels, ils peuvent procéder à sa réduction et à produire des logiciels économes en énergie. Ainsi, cette thèse apporte 3 contributions centrées sur le langage Java. La première contribution vise à aider les développeurs à choisir et à configurer leur environnements d'exécution. Nous avons identifié d'importantes différences dans la consommation d'énergie en utilisant différentes plateformes JVM avec différentes configurations JIT et GC. Les deuxième et troisième contributions étudient l'impact sur la consommation d'énergie de petits changements que les développeurs appliquent souvent sur leur code source (refactoring de code et remplacement d'API/méthodes respectivement). Nous montrons à travers ces études que le refactoring structurel du code source n'impacte pas la consommation énergétique des logiciels de manière considérable. En revanche, le remplacement des méthodes de lecture/écriture dans Java impacte considérablement la consommation d'énergie selon le cas d'utilisation.

Cette thèse contribue à enrichir les connaissances sur la conception de logiciels verts et fournit des approches et des conclusions pour améliorer l'efficacité énergétique à plusieurs niveaux du développement de logiciels.

Table of contents

List of figures

List of tables

Nomenclature

Acronyms / Abbreviations

CI/CD	**C**ontinious **I**ntegration / **C**ontinious **D**elivery
CLBG	**C**omputer **L**anguage **B**enchmarks **G**ame
CPU	**C**entral **P**rocessing **U**nit
G5K	**G**rid 5000
GC	**G**arbage **C**ollector
GSD	**G**reen **S**oftware **D**esign
I/O	**I**nput / **O**utput
ICT	**I**nformation (and) **C**ommunications **T**echnology
JDK	**J**ava **D**evelopment **K**it
JIT	**J**ust-**I**n-**T**time compiler
JMH	**J**ava **M**icrobenchmark **H**arness
JRE	**J**ava **R**untime **E**nvironment
JVM	**J**ava **V**irtual **M**achine
KPI	**K**ey **P**erformance **I**ndicator
OS	**O**perating **S**ystem
OSS	**O**pen **S**ource **S**oftware
PUE	**P**ower **U**sage **E**ffectiveness
RAPL	**R**unning **A**verage **P**ower **L**imit
SEC	**S**oftware **E**nergy **C**onsumption

STD	**ST**andard **D**eviation
VM	**V**irtual **M**achine

Chapter 1

Introduction

Modern times and lifestyle created numerous needs and usages, especially in the digital sector: among others, one could easily mention connected devices, wearables, Internet of Things, smartphones, tablets, etc. At the same time, many existing services have migrated, at least partially and sometime even totally, their activities to the Internet : online retailers, banking, advertising, video and music consumption and even public services. All these new activities have increased the overall environmental footprint of the Information and Communication Technology (ICT) sector, which is estimated to be responsible for approximately 4% of the greenhouse gaz (GHG) emissions worldwide in 2020 with a worrying 8% growth rate, according to the French think tank *The Shift Project* [137], or 2% according to [13], a similar number to the aviation sector contribution.

Energy consumption is one of the reasons for this environmental footprint, and it has quite logically increased as well. Consumers devices (household equipments, computers, mobile devices, etc.) are responsible for some part of this increase, while the networks, especially mobile radio networks, also share a large responsibility. For data-centers, however, which provide the storage and computing power required by all these services, things are not very clear ; the evolution of their energy consumption is still subject to debate today. While some assume a significant increase in the energy requirements of energy data centers [55], others [86] argue that their energy consumption has been stable in the last 10 years, even though storage and computing capacities have known a tremendous increase during the same period, thanks to the improvements that have been made to achieve better energy efficiency in these infrastructures.

In any case, one can easily argue that stable consumption is not really such a good news, when the objective is to actually reduce the global footprint of every sector, including the ICT one. There is also a fear that energy efficiency improvements of the last years will not be sustained in the future, as the *"low hanging fruits"* have already been harvested, and that the continued increase in computing power might not be offset in the coming years.

Ultimately, the purpose is to reduce the digital energy consumption and/or its increasing cadence. For data centers, multiple actions can be discussed in order to improve their energy efficiency. These actions can take place at multiple stages in data centers' life cycle. During the acquisition phase, one can limit the purchase of components to the minimum needed, choosing components with low power consumption and a least consuming manufacturing process when possible. During the transportation and installation phase, the acquisition of components can be grouped together in order to reduce the transportation CO2 cost, for example. In the usage phase, multiple solutions are possible: using green energy sources, lowering the energy cost of cooling, prioritizing repair to replacement, etc. Finally, the end-of-life recycling phase can also be tuned to be energy efficient. However, existing Life Cycle Assessments demonstrate that for data centers [151], the usage phase is responsible for the vast majority of their environmental impact, mainly due to the electricity consumption. Therefore, reducing that energy consumption is one of the most important factors to reduce the overall environmental footprint of data centers.

Most current and past work on data center energy efficiency have focused on the infrastructure, the hardware, and its optimization. This includes, for example, reducing the energy consumption of servers, cooling systems and other hardware components that are essential for the proper functioning of the data center. These improvements are reflected in substantial progress made on the Power Usage Efficiency (PUE) [95], an indicator commonly used to measure how efficiently a data center uses energy.

However, as it is based on server energy consumption, the PUE only measures the efficiency of the data center *infrastructure*; it does not take into account the efficiency of the servers themselves, or more precisely, of the software running in these servers. Indeed, a modern data center running only highly inefficient and badly written software can still have a very good PUE. Thus, the software part should actually be considered as important as hardware and infrastructure. In fact, the software running within servers and the data volumes transiting between storage bays, switches and routers are responsible for using the hardware resources. This usage can for example increase the clock frequency of the hardware and/or increase the temperature of the data center, which requires more energy for the hardware components and cooling.

Reducing software energy consumption (SEC) is thus very important to enhance the energy efficiency of a data center and its CO2 impact. Numerous researches provided studies to reduce SEC [35, 62]. These studies focused on multiple aspects of the assessment of software energy, such as the accuracy and granularity of the measurements [53, 150], or the different ways of reducing SEC [52, 126], for example.

The main purpose of our thesis is to help developers to produce software that consumes less energy. We believe that promoting GSD (Green Software Design) and SEC considerations should go through an important educational phase, to ensure that the community in general and developers, in particular, are well motivated and aware of the stakes of GSD and their prominent role in reducing SEC and building less consuming software. This is easier said than done, as there is not enough knowledge on how to build "green" software that can carry developers' choices for every use case. However, the man who moves a mountain begins by carrying away small stones. Small enhancements and insights in this topic are still very welcome to constitute a broad and robust set of knowledge that can be used to reduce SEC.

The remainder of this chapter is as follows. Section 1.1 states the main problems of green software design. Section 1.2 expresses the objectives of this thesis. Finally, Section 1.3 explains the organization of this document, including some contributions of the main chapters.

1.1 Problem Statement

The problem of digital energy consumption is critical. Numerous research and estimations highlighted the urgent situation that needs to be truly considered. In fact, future ICT infrastructures can hardly slow their overall electricity use until 2030, and they will keep on using more energy than today, despite the decrease of the energy cost per operation [8, 9]. Moreover, data centers are estimated to account for around 1% of worldwide electricity use [95].

Although the PUE for data centers and the cost per computation / terabyte storage tends to decrease [86, 95], the service demand is estimated to largely increase, including the global data centers traffic, the global storage capacity, servers count, and data centers workload [95].

Similarly to hardware, software energy consumption plays a crucial role in the global energy consumption of data centers. It is however a very complex problem to identify the hotspots of software and reduce its energy consumption. SEC is mostly considered as a young topic. Many researchers have been actively contributing during the last decade to tackle these issues and disseminate the considerations of green software design among users, developers, and other entities. However, the current situation is still not mature enough to provide concrete solutions for multiple use cases, and guide developers to reduce the energy consumption of the produced software.

Ultimately, SEC should have a similar importance and significance as software performance. This is, however, far from being the case today. One major reason is that software performance was prioritized for a long time. Seeking fast software and reducing the execution/response time was always requested, as the results were immediately perceived. On the other hand, the energy resource was not considered to be as critical as execution time for quite a while. Now that the myth of infinite resources is not really arguable, and even with renewable energy sources, developers should put more attention to the energy efficiency of what they are producing.

One major problem next to the lack of knowledge is the bad communication and popularization of the acquired knowledge. In fact, many studies results are not intended for developers. This implies that no mature tool is made available to developers to assist them in reducing the energy consumption of their software as it is the case for performance (such as Sonarqube, for example).[1]

In this thesis, we try to tackle some of the previous problems, by first understanding developers' needs. Then, by extracting concrete guidelines from the studies/contributions we conduct during the thesis.

1.2 Objectives

Our main goal is to help developers to reduce the energy consumption of the developed software. To approach this, we constitute a set of sub-objectives that will drive our thesis.

The first objective of this thesis is to cover developers' understanding and awareness of GSD. This includes their sensitivity to the topic, their willingness to improve the current situation and what they require to achieve it. Concerning the last point, we aim at identifying the hurdles that developers are facing against a proper consideration of SEC, their tooling needs, as well as ways to broadly improve and promote GSD within companies.

Next, we aim at delivering some insights on software energy consumption for Java and Java Virtual Machine (JVM). First, we intend to investigate how to conduct robust and reproducible experiments. For this, we need to ensure the accuracy and steadiness of the energy measurements.

Then, we evaluate the behavior of different execution environments in terms of energy consumption. The purpose is to investigate whether changing or tuning the execution environment—particularly the Java environment—can substantially alter SEC.

Finally, we also aim at delivering insights and guidelines for Java GSD at source code level. Especially the impact of some minor changes that developers often apply on their source code, such as code refactoring or libraries substitution.

Concretely, we want to answer the following research questions:

RQ 1: How do developers perceive GSD? what are the requirements to promote its consideration?

RQ 2: How to conduct robust, steady and accurate energy consumption measurements?

RQ 3: What is the impact of the JVM choice/configuration on SEC?

RQ 4: Does code refactoring have a substantial impact on the evolution of SEC?

RQ 5: Do minor changes on source code such as methods or libraries refactoring have a substantial impact on SEC?

[1]https://www.sonarqube.org/

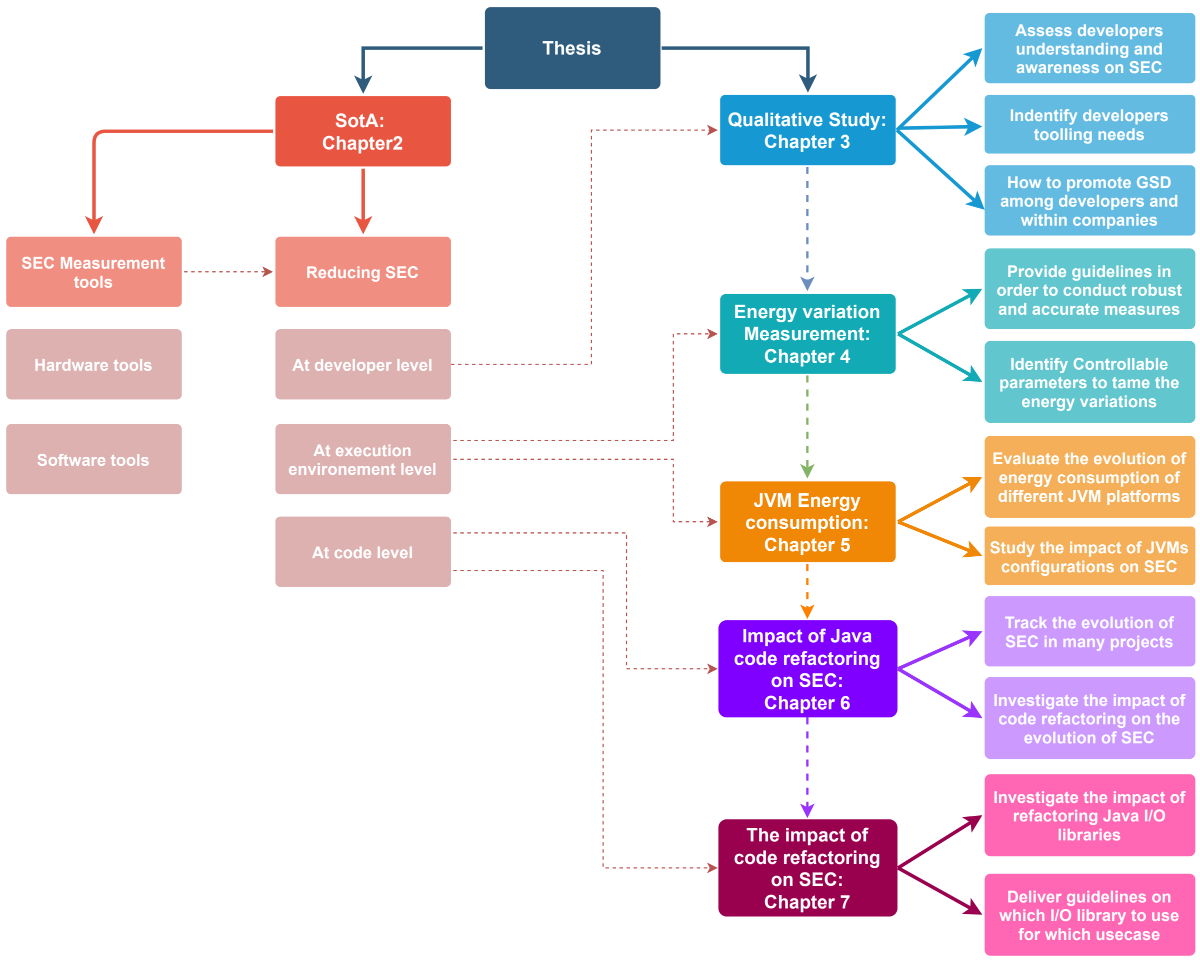

Thesis
SotA: Chapter2
Qualitative Study: Chapter 3
SEC Measurement tools
Reducing SEC
Hardware tools
At developer level
Software tools
At execution environement level
At code level
Assess developers understanding and awareness on SEC
Indentify developers toolling needs
How to promote GSD among developers and within companies
Energy variation Measurement: Chapter 4
Provide guidelines in order to conduct robust and accurate measures
Identify Controllable parameters to tame the energy variations
JVM Energy consumption: Chapter 5
Evaluate the evolution of energy consumption of different JVM platforms
Study the impact of JVMs configurations on SEC
Impact of Java code refactoring on SEC: Chapter 6
Track the evolution of SEC in many projects
Investigate the impact of code refactoring on the evolution of SEC
The impact of code refactoring on SEC: Chapter 7
Investigate the impact of refactoring Java I/O libraries
Deliver guidelines on which I/O library to use for which usecase

Figure 1.1: The organization of the different chapters of the document

1.3 Organization

Figure 1.1 depicts the organization of the chapters. The remainder of the document can be summarized as follows.

- **Chapter 2** summarizes the state of the art on software energy consumption. It introduces numerous hardware and software tools used in the literature to measure SEC. Then, it discusses developers understanding and awareness of SEC. Finally, the chapter presents solutions of some studies aiming at improving software energy efficiency at execution environment level and source code level.

- **Chapter 3** investigates developers' knowledge and awareness on SEC and highlights the motivation of our work regarding the immaturity of the topic. The purpose of the chapter is to identify developers' requirements in terms of green software design (GSD) tooling, but also how to promote GSD within companies and among developers. This chapter stipulates implications for developers, decision makers, tool creators, and researchers.

- **Chapter 4** is motivated by the lack of convenient tools that was reported in the previous chapter. Energy measurement being the first brick of GSD tools, this chapter shows how energy measurements can vary and deliver inconsistent results for the same job, executed on similar nodes or even on the same node. This chapter also provides guidelines on controllable factors that practitioners can easily tune to tame this variation and conduct more robust and stable energy measurement experiments, which will substantially benefit the next chapters studies.

- **Chapter 5** introduces one of the runtime platforms adopted by developers, "the JVM", and focuses —similarly to the next chapters—on Java. It studies the impact that the choice of a JVM platform and its configuration could have on the energy consumption of software. Concretely, it investigates the differences in terms of energy consumption between multiple versions of JVMs of several constructors, along with the JIT and GC configurations that could reduce the SEC. It delivers insightful advices on how to set up the JVM execution environment to reduce the SEC before deployment.

- **Chapter 6** represents a study of SEC at code level. The purpose of this chapter is to track the evolution of SEC of some Github projects and check whether structural code refactorings can enhance or deteriorate the energy consumption of software over the years. The conclusions of the study show that the SEC of projects functionalities trends to decrease over time, and that structural code refactorings have a very mitigated impact on SEC and could thus be applied without substantial drawbacks.

- **Chapter 7** more specifically evaluates the impact that the choice of the Java I/O library could have on SEC. 27 different I/O methods and multiple scenarios and use cases have been considered to evaluate the most/least consuming ones. Refactoring the default I/O methods of real Java projects/libraries proved to be energy efficient, with up to 30% of savings in energy consumption.

- **Chapter 8** presents our concluding remarks, contributions, and perspectives.

The main chapters of this thesis have been published in major software engineering conferences as full papers: ESEM'20, ESEM'21, and ICSME'21. Concretely, Chapter 3 has been published in ESEM 2020 [109]. Chapter 4 and Chapter 5, in collaboration with another thesis, are published as ICPE 2020 [108] and ESEM 2021 [110] papers, respectively. Chapter 6 has been published in ICSOFT 2021 [112], while chapter 7 has been accepted at ICSME 2021 [111]. These chapters have been revised and updated when writing this thesis.

Chapter 2

State of the Art on Software Energy Consumption

Energy consumption efficiency is a well-known concept. In most domains, the purpose is to reduce the consumption of the electronic devices/parts. Modern times even witness energy classification (A, B . . . F) for many electric and electronic devices, such as screens and household appliances, to give the consumer an idea of the energy consumption of his devices, which will reflect on his electricity bill afterwards.

In computer science, the purpose is pretty much the same. Many researches have been carried out on energy optimization. Some of these works are focusing on reducing the energy consumption at the hardware level, while others focus on optimizing software's consumption.

Figure 2.1 depicts the distribution of the digital energy consumption in 2017 [8] ("p" for production and "u" for usage). It shows that the impact of both production and usage phases are extensive. Moreover, the usage costs of data centers, networks, and terminals are all significant (19%, 16% and 20% respectively).

For data centers, Avgerinou et al. [13] studied the evolution of power usage effectiveness (PUE) for some companies participating in the European code of conduct for data center energy efficiency program. The study reported on a slow decrease in the PUE of data centers which represents the ratio of the total facility energy to the IT equipment energy. A low PUE indicates that most of the energy is used for data centers IT equipment, and only little energy is used for other purposes such as cooling and lights.

In this thesis, we focus on the energy efficiency induced by the software part rather than hardware. Green software design can be defined as the energy consumption induced from software lifecycle, including the planning, design, development, installation, use, and decommissioning phases. The purpose of GSD is the development of a manageable software that meets the present needs in a defined context, fulfilling a function over a time span with the minimum environmental and ecological impact.

Reducing SEC is a part of GSD that focuses on the usage phase of a software.

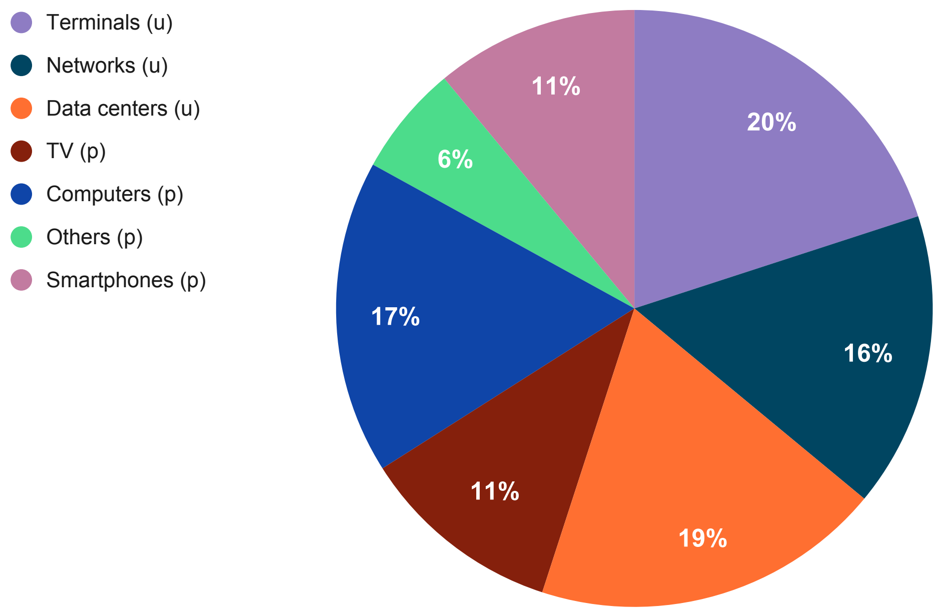


Figure 2.1: Distribution of the digital energy consumption in 2017

The second decade of the 21th century knew a recognition of software energy consumption as a key topic by some researchers and entities [119, 121].

This chapter aims to summarize the state of the art on SEC by highlighting some of the progress achieved in the topic. The first major requirement towards reducing SEC is to be able to accurately measure the consumed energy. Section 2.1 reports thus on software energy measurements. It includes examples of software and hardware measurement tools and indicates the differences, advantages and disadvantages of these tools. It also discusses the causes of energy measures variations, that constitutes a key challenge to reason upon accurate measurements.

Section 2.2 discusses studies that cover developers and users' needs and understanding of SEC, and how to sensitize them about the importance of such considerations. In fact, such knowledge is very important to collect developers requirements and needs, which can be used to guide and shape further work. Finally, Section 2.3 reports on contributions to reduce SEC at **execution environment level** and **source code level**. Studies at the execution environment level help developers configuring and optimizing their infrastructure and execution environment, so their software consume less energy (with a focus on the JVM platform). Studies at the code level, on the other hand, concern optimizations and changes on the source code of the software itself to reduce its energy consumption. Including frequent changes that developers often apply on the source code such as code refactoring.

2.1 Measuring Software Energy Consumption

Achieving software energy efficiency is an iterative process that consists in many enhancements and improvements to obtain a better source code and/or better execution environment without impacting software functionalities, evolution and maintainability.

In order to produce an energy-efficient software, one first needs to faithfully assess software energy consumption and record its evolution over time.

Numerous tools have been presented in the literature to measure software energy consumption. One can distinguish 2 categories of tools: hardware tools and software tools.

2.1.1 Hardware tools

This category of tools requires a dedicated hardware to measure the energy consumption, generally known to be very precise, but not fine-grained (give the general EC) and incur an instrumentation phase, possibly an additional financial cost. In most cases, they are digital measurement devices/boards that assess the energy consumed from the power supply in joules or Wh. Similar tools have been used in the literature to measure software energy consumption. The way these tools are used consists of measuring the energy consumption of a computer or a server before and during the software's execution. Software energy consumption is thus inferred by subtracting the extra energy consumption that was induced by the software.

First, WattsUp Pro [56] is a device that has been used in several works [62, 96]. It is usually installed between each computer/server and its power source to record the total energy consumption at a maximum sampling rate of 1 sample per second and a maximum current of 15 amperes. The device has an internal memory that can be used to log the measures. The measures can cover a wide variety of data, including maximum potential, current, power, and cumulative costs. The memory-stored data can be downloaded to a computer via a USB cable. The product also works with a separate computer software Logger Pro or LabQuest App to create graphs, calculations, and device profiles. If the need is to measure the energy consumption of a cluster simultaneously, many WattsUp Pro devices might be necessary. The price of a WattsUp Pro device fluctuates between 50$ and 120$.[1] It can thus be very expensive for a large project to acquire a set of devices.

PowerMon [20] and PowersMon2 [19] are other examples of hardware energy measurement boards. These power monitoring devices operate between a system's power supply and a motherboard, to analyze the power consumption tradeoffs in software and computer applications. PowerMon monitors voltage and current on six DC rails and reports measurements at a rate of up to 50 samples per second through a USB interface, allowing monitoring by the target host or a separate host. PowerMon2 is a bit smaller, compared to its predecessor, allowing it to be used in a 1U server chassis in the same 3.5 inch hard drive form factor. It also has 2 more

[1]https://camelcamelcamel.com/product/B000CSWW92

DC rails (for a total of 8) for additional peripherals, such as disks and GPUs. PowerMon2 also uses a USB cable to report the power measurements at a maximum rate of 3 KHz. The price of a single PowerMon device is higher than 150$.[2]

Similarly to PowerMon devices, PowerInsight [75] is another tool built on an external board based on an ARM Cortex Processor. It is designed by Penguin Computing to accomplish component-level power and energy instrumentation of commodity hardware. PowerInsight can be connected to up to 15 components (disks, GPUs, etc.). It is also designed to work within a cluster.

Each board inserted between a computer's motherboard and its power supply is equipped with an Ethernet port. This port is used to send and acquire data from/to the main node that aggregates and saves data for the cluster's energy consumption. Measurements have been tested with the PowerInsight tool with a sampling rate of 1 sample per second.

GreenMiner [54] is a well-known measurement tool used in the literature, mainly to measure the energy consumption of mobile applications [52, 134]. It is a hardware/software test suite that runs tests on numerous devices and measures the energy consumption and the power usage of the entire device. The GreenMiner client side is a Raspberry Pi that acts as a testbed. It uses an Android test device to run the tests and collects the results from an Arduino board that monitors the energy consumption. Energy is measured via an INA219 energy measurement chip that samples and aggregates measurements with a frequency of 5 MHz. The testbed records and uploads the INA219 measurements to the GreenMiner web service that represents the server side, responsible for treating and analyzing the energy measurement data.

Other works have also used some other hardware tools to measure the software energy consumption in their studies, such as a 1500$ analog-to-digital data acquisition (DAQ) card (National Instruments USB-6215) that samples the amount of power consumed by the component at a 10 KHz frequency [90].

On-Chip Power Sensors

On-Chip power sensors constitute a sub-category of hardware measurement tools. These tools are generally integrated with the underlying hardware and does not require extra boards or devices.

RAPL [37, 67, 68] (Running Average Power Limit) is an Intel measurement tool. It is the most commonly used tool to measure software energy consumption. Since sandy-bridge mirco-architecture, Intel CPUs (but also AMD CPUs since family 17h Zen)[3]) use dedicated RAPL registries to retrieve the power consumption every millisecond [58]. As illustrated in Figure 2.2, RAPL is capable of delivering the energy consumption of the CPU package,

[2]https://renci.org/technical-reports/tr-09-04/
[3]http://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html

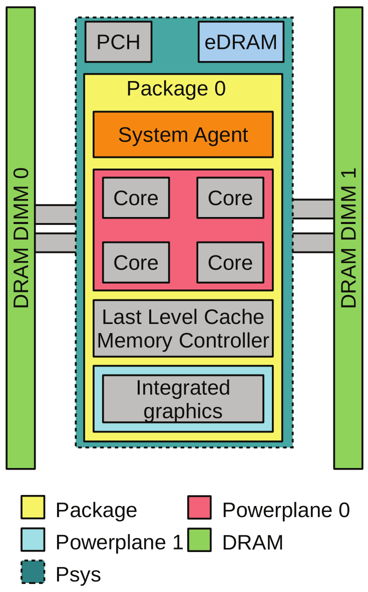


Figure 2.2: Power domains supported by RAPL [67]

constituted of CPU Core (cores and cache) and the CPU Uncore (LLC, integrated GPU, etc.), in addition to DRAM. This high accuracy tool [67] is very easy to use as it requires no hardware modification or extra purchases, as it is integrated within most of the latest Intel and AMD newest CPU. RAPL also supports power capping to limit CPU energy consumption.

Similarly to RAPL, Nvidia Tesla GPU are equipped with power sensors [27]. These sensors can be queried through Nvidia Management Library (NVML), which is a C-based programmatic interface for monitoring and managing various states within NVIDIA Tesla GPUs [41].

2.1.2 Software tools

Software-based measurement tools are not devices, but software plugins and sensors (usually cost-free), based on some other hardware tools to assess the energy consumption. The main purpose behind these tools is granularity. In fact, most hardware tools only give the global energy consumption of the whole system/component (computer, server, motherboard, etc.). The measures are, however, less accurate than hardware tools as they are often built over empirical estimations and data learning mechanisms. Many software measurement tools learn the behavior of a power model and deliver energy consumption estimations. This model is then used to distribute the observed energy consumption, via a hardware tool, between several execution entities, at process level, control group, thread, or even code line.

PowerAPI [32] and SmartWatts formula [43] are first examples of software measurement tools. These tools capture global energy consumption measures from RAPL and use other system events such as cache misses/hits and CPU frequencies evolution (DVFS) through a sensor, to build a power model of the control groups (system control groups, docker containers, kubernetes pods, etc.) based on a Ridge regression. The model continuously learns and enhances energy consumption data in real-time with a maximum frequency of to 100hz. The tool has a lightweight decentralized design. Only the lightweight sensors are installed within the monitored machines to capture and send data to the main server. The SmartWatts formula is then run on the main server to build the model that allows assigning the energy consumption for each running control group. PowerAPI only works on Linux, installed on a bare-metal physical machine.

WattWatcher [77] is a multi-core power measurement framework that offers fine-grained energy measurements at process level. This tool is based on power models to estimate the energy consumption of processes. To build the power model, CPU events are forwarded from the measured node to a model generator node. It operates by passing event counts and a hardware descriptor file into a configurable back-end In fact, WattWatcher uses a set of predefined CPU architectures descriptors. This requires users to have a deep knowledge of their hardware architecture and provide it by filling the hardware configuration file. The tool uses several calibration phases to build a robust model. The authors claim an error margin lower than 3% for the energy consumption measures.

Joulemeter [62, 71] is a windows based software provided by Microsoft that uses power models (for CPU, memory and disks) to estimate the energy consumption of windows running applications down to process level. It uses low-overhead power models to infer power consumption from resource usage at runtime, and offers power capping capabilities for VMs. Previous experiments with Joulemeter [61] showed that the tool provides a general idea of energy consumption that differs significantly from the real energy consumption. Joulemeter first needs a calibrating phase to adapt its models to the hardware it runs on. Only 1 process can be measured per instance of Joulemeter with a sampling rate of one sample per second.

JRAPL is another example of SEC estimation tool used in numerous works [49, 83, 122]. This Java framework allows profiling and measuring the energy consumption of Java applications, functions or even a set of code lines. The measures are highly based on the RAPL provided data. Thus, the global energy consumption obtained through RAPL between two timestamps (start and end of the code to measure) is used to deduce the energy consumption of the Java code. Experiments using JRAPL should be run on a well-configured system to reduce the overhead of the OS and user processes on the global energy consumption used by JRAPL. Pyjoules[4] offers similar utilities to JRAPL, but for Python. This framework is also based on RAPL's global energy consumption, with an integration with Python source code to measure the energy consumption of a code snippet.

Jolinar [103] is another process-level energy consumption measurement tool. The tool needs no calibration phase and uses pre-established power models built on some hardware parameters (TDP, disks I/O rate, etc.). These parameters should be found and provided by the

[4]https://pyjoules.readthedocs.io/en/latest/

user for his hardware. Jolinar can only measure the energy consumption of 1 application at a time, and only measures the energy consumption of its main process. The results are displayed at the end of the execution, they include the energy consumption of CPU, DRAM, and disks.

For a Java application, Jalen [102] is able to profile and measure the energy consumption down to the method level. Jalen can use code instrumentation and statistical sampling at a specific rate to collect data. The authors advise using the second method due to the overhead that code instrumentation could add. Jalen captures the JVM's stack trace with the CPU time of threads each 10 ms and computes statistics about the method calls. These statistics are then used to deduce the energy consumption of each method.

2.1.3 Energy Consumption Variations

By energy variations, we refer to the problem of steadiness and stability when measuring software energy consumption. This means that ruining the exact same job or application within the exact same environment and configuration can result in different measures in energy consumption due to many factors [53].

This variation has often been related to the manufacturing process [31] of CPU and components, but has also been a subject of many studies, considering several aspects that could impact and vary the energy consumption across executions. The correlation between the processor temperature and the energy consumption was one of the most explored paths. Joakim v Kisroski et al. reported that identical processors can exhibit significant energy consumption variation with no close correlation with the processor temperature and performance [63]. The registered differences in energy consumption were most significant for idle and high load states with up to 29.6% and 19.5% variations respectively. However, Wang et al. [150] claimed that the processor thermal effect is one of the most contributing factors to the energy variation, and that the correlation between the CPU temperature and the energy consumption variation is very tight. The Spearman correlation coefficient between CPU temperature and energy consumption was higher than 0.93 for all experiments. This makes the processor temperature a delicate factor to consider when comparing energy consumption variations.

The ambient temperature was also discussed in other papers as a candidate factor for the energy variation of a processor. Varsamopoulos et al. [143] claimed that energy consumption may vary due to fluctuations caused by the external environment. These fluctuations may alter the processor temperature and its energy consumption. However, the temperature inside a data center did not show major variations from one node to another.

Diouri et al. [38], the study showed that switching the spot of two servers does not substantially affect their energy consumption (this was also confirmed by Wang et al. [150], where the rack placement and power supply only introduced a maximum of 2.8 % variation on the observed energy consumption). Moreover, changing hardware components, such as the hard drive, the memory or even the power supply, does not affect the energy variation of a node, making it mainly related to the processor. The authors also noticed during their experiments that old CPU versions could constitute a cause for a higher variation.

Beyond hardware components, the accuracy of power meters has also been questioned. Inadomi et al. [59] used three different power measurement tools: RAPL, Power Insight and BlueGene/Q EMON. The study showed that the variations in energy consumption are not related to energy monitoring tools. In fact, all of the three tools recorded the same 10% of energy variation on the executed benchmarks. The authors related those variations mainly to the manufacturing process.

Mitigating Energy Variations. By acknowledging the energy variation problem on processors, some papers proposed contributions to reduce and mitigate this variation.

Inadomi et al. [59] introduced a low-cost scalable variation-aware algorithm that improves application performance under a power constraint, by determining module-level (individual processor and associated DRAM) power allocation. The algorithm uses a model to predict the power consumption of applications. The model is based on the assumption that energy consumption for both CPU and DRAM is proportional to the CPU frequency. Experiments recorded up to 5.4× speedup on a 1920 sockets environment.

Acun et al. [4] suggested a way to reduce the energy variation on Ivy Bridge and Sandy Bridge processors, by disabling the Turbo Boost feature to stabilize the execution time over a set of processors. They also formalized some guidelines to reduce this variation by replacing the old/slow chips with recent ones, by load balancing the workload on the CPU cores. They also claimed that the variation between the CPU cores is insignificant.

Chasapis et al. [28] introduced an algorithm for parallel systems that can be used to reduce the energy variation by compensating the uneven effects of power capping. The algorithm needs a calibration phase, its approach considers numerous execution segments of the application parallel execution and associates each segment with a particular power and number of active core assignations per socket. It starts with evenly distributing power and activating all cores and then progressively iterates over a set of available power/#cores configurations and selects the best one. For each configuration, the targeted application runs for a certain amount of time called a monitoring window.

Contrary to some paper results [38], Marathe et al. [93] highlighted the increase of energy variation across some recent Intel micro-architectures by a factor of 4 from Sandy Bridge to Broadwell. Moreover, they noticed a 15% of run-to-run variation within the same processor and the increase of the inter-cores variation from 2.5% to 5% due to hardware-enforced constraints. The authors suggested some recommendations for Broadwell chips to reduce the energy consumption variations, such as leaving one hyper-thread per core idle for the system processes, or avoiding co-locating high and low efficiency processors on the same node on large-scale clusters.

Tool	Kind	Granularity	Sampling Rate	Highlight
WattsUp Pro	Hw	System	1 sample/sec	Pricey, USB connectivity
PowerMon	Hw	System	50 samples/sec	Pricey, up to 6 plugged components
PowerMon2	Hw	System	3000 samples/sec	Pricey, up to 8 plugged components
PowerInsight	Hw	System	50 samples/sec	Up to 15 plugged components, cluster compatible
WattProf	HW	System	-	-
GreenMiner	Hw/Sw	System	50 samples/sec	Designed for Android applications
RAPL	Hw	System	1000 samples/sec	Core, Uncore and DRAM energy consumption
SmartWatts	Sw	C-group	100 samples/sec	Core, client-Server architecture, power estimation model
WattWatcher	Sw	Process	-	Several calibration phases, requires a Hw descriptor file
Joulemeter	SW	Process	1 sample/sec	For Windows, one process at a time
JRAPL	SW	Method	1 sample/sec	Java code profiling, based on RAPL
Pyjoules	SW	Method	1 sample/sec	Python code profiling, based on RAPL
Jolinar	SW	Process	1 sample/sec	One process at a time
Jalen	SW	Method	100 sample/sec	Java code profiling

Table 2.1: Energy measurement tools

2.1.4 Summary

As a summary of software energy measurement tools, we distinguish two categories of those tools. First, hardware tools. They require specific or additional hardware to measure the energy consumption. The measures are precise but very often global of the whole measured system. These kinds of tools cannot be used when targeting a fine-grained energy assessment. Software tools on the other hand come on top of hardware tools to provide fine-grained energy measurement (processes, C-groups, threads, methods,etc.). These tools are usually not completely independent and use some measurements or events provided from the hardware. Mostly built on estimation models, these tools are not as precise as hardware tools for energy measurements. Fahad et al. [41] conducted a comparison between some of these hardware tools(external: WattsUp Pro and on-chip: RAPL) and model based software tools (Extrae [6] and Paraver [92]). They highlighted that these tools can report significant differences in energy measurements. Table 2.1 lists and summarizes some of the tools described previously.

Many studies showed that software energy measurements are not stable. Numerous papers approached the problem of SEC variations and tried to identify the causes behind them, such as components manufacturing process, temperature, age, etc. These variations can be up to 30%, and can be recorded between identical nodes, and even within the same node. They should thus be carefully considered when conducting SEC measurements.

2.2 Understanding Software Energy Consumption

In order to promote green software design and reduce software energy consumption, it is important to understand the current situation, including developers and users needs on one hand, and sensitize/teach them about the subject on the other hand. In the past decade, some researchers have started such investigations to deliver an understanding on how to enhance practitioners experience on GSD, especially within companies.

First, Pang et al. [114] surveyed 100 persons. They focused on investigating programmers' knowledge of software energy consumption. Their results expressed a lack of knowledge on ways and best practices to reduce software energy consumption, especially for desktop computers compared to mobile devices. The paper claims that only 10% of the participants try to measure the energy consumption of their software project, and only 17% consider reducing software energy consumption as a requirement in software development. Moreover, the authors mention an urgent need for training and education on software energy efficiency.

Pinto et al. presented an empirical study on how programmers understand software energy consumption problems [119]. The authors used more than 300 questions and 550 answers from 800 participants on *Stack Overflow* (SO) [5] as their primary data source. They stated that practitioners are aware of software energy consumption problems, but have limited knowledge and vague answers on how to deal with it. The extracted knowledge was divided into 5 themes: energy measurement, general knowledge, code design, context specific and noise. The authors also summarized 7 major causes and 8 common solutions evoked by developers for software energy consumption problems. One of the encouraging insights of the paper is the yearly increase of GSD-related questions and answers on SO, with a peak of 180%. 85% of these questions are answered, 45% are correctly answered with an average of 2.6 answers per question.

In a later work, Manotas et al. [91] conducted a mixed quantitative and qualitative study, applied on 464 candidates from ABB, Google, IBM and Microsoft, and 18 from Microsoft respectively. The study starts with the qualitative study. It consists of 18 semi-structured interviews of 30-60 minutes each in order to deeply explore and understand participants' opinions. The interviews were recorded, transcribed and then analyzed using open, axial and selective coding. Then, a quantitative study is conducted with a larger set of participants in order to quantitatively assess the quality of information learned from the interviews.

The study provided some interesting results, reporting for example that: *i)* mobile developers are more concerned about software energy consumption problems, *ii)* energy concerns are largely ignored during maintenance, *iii)* energy requirements are more often desires rather than specific targets, *iv)* developers believe they do not have accurate intuitions about the energy usage of their code and are undecided about whether energy issues are more difficult to fix than performance issues, and *v)* 93% of the survey participants want to learn about energy issues from profiling and static code analysis.

Interestingly, this last result is in contrast to Johnson et al.'s work [64], in which they interviewed 20 candidates through a qualitative study. They highlighted that developers do

[5]https://stackoverflow.com/

not use static code analysis to find bugs, which adds a question mark on why are developers more open to static code analysis when it comes to energy considerations.

In another similar work [62], the authors conducted a set of semi-structured interviews to discuss the added value of an applied energy profiling method across releases of software. More specifically, this work compares two versions of the same software and interviews the developers while highlighting the registered increase in the energy consumption of the later version after specific software changes (add of an encryption module). It discusses how such a quantification of software energy consumption helps developers to create awareness and eventually consider energy efficiency aspects when planning software releases.

Nevertheless, only few studies investigated the hurdles against SEC considerations within companies and among developers [114]. Furthermore, to the best of our knowledge, no study investigated developer's requirements and needs in terms of tooling to efficiently track and reduce software energy consumption.

2.3 Reducing Software Energy Consumption

Even if the topic of SEC is not mature yet, it has become an important topic and the subject of many researches. The last decade witnessed an emergence of several papers and studies about software energy efficiency. For the rest of the section, we showcase many of these studies, organized at 2 different levels.

2.3.1 At Execution Environment Level

Several works have been conducted to improve the energy efficiency of the infrastructure and the execution environment. The purpose is to set up and configure an environment where software runs with the least energy consumption. This might include infrastructure sizing, virtual machines and processes allocation/placement, system configuration, software execution and scheduling, etc.

For instance, many works have been pursued to extend the battery life and reduce the energy consumption of mobile applications. Most of these works only focus on achieving battery life savings, which does not completely reflect the energy efficiency of an application, due to the high network and back-end cost that it induces. Balasubramanian et al. [15] compared the energy consumption of 3 networking technologies (3G, GSM, and WiFi). They realized through comparison that 3G and GSM consume much more energy than WiFi. They also suggested a protocol TailEnder that reduces SEC by re-scheduling transfers for applications and packages that accept some delay (such as emails), achieving up to 2 times energy consumption improvement.

In another example, Othman and Hailes [106] showed through a simulation that users jobs can be transferred from a mobile host to a fixed host (method offloading) to reduce the energy consumption of the mobile application and extend the battery lifetime by up to 20%. This offloading method reduced both the energy consumption on the mobile devices and the response time in some cases as the methods are sent to faster servers with higher performance. However, the global energy consumption when using methods offloading does increase as it includes the energy consumption of the offloaded methods on the servers and network energy cost.

Other than mobile applications, Ribic and Liu presented Aequitas [126]. Aequitas is a framework that helps to co-exist parallel applications on co-managed power domains, i.e. that share the same resources (CPU cores). The applications cohabitation consists of a energy-performance trade-off. Aequitas mainly achieves its purpose with a round-robin algorithm (or other contention policy such as first-come-first-serve, or a policy to average the CPU frequencies requested by contending parties) that allows the different applications and their sub-threads to access the underlying hardware power management within their share of time, with a focus on energy consumption. The experiments reported on up to 12.9% of potential energy saving against only 2.5% of performance loss.

The energy consumption of VMs allocation and tasks placement has also been studied [97]. The authors suggest an algorithm to map tasks to VMs and VMs to physical machines in an energy efficient way. Based on the resources requirements of each task, the algorithm selects a VM then a physical machine where the VM can be deployed. The authors claimed that using their allocation algorithm ETVMC reduces the number of active physical machines along with the task rejection rate using a cloud simulator.

In another study on VMs, Kurpicz et al. discussed the total energy consumption of a VM in a data center while highlighting the static cost [73]. They presented their model *EPAVE* as being transparent, reproducible and predictive cost calculator for VM based environments. *EPAVE*'s role is to attribute data center's static and dynamic costs to each VM. The dynamic cost includes the dynamic energy consumption part of the servers, routers and storage devices, while the static cost aggregates the idle consumption of nodes and routers, air conditioning, power distribution, etc.

According to Eddie Antonio Santos et al. [39], the usage of docker containers to run software does not add a substantial overhead to the energy consumption. Concretely, this empirical study compared the energy consumption of bare-metal applications against docker containerized ones. The results reported on a non-substantial difference in energy consumption except for docker I/O system calls. They advised developers worrying about I/O overhead to consider bare-metal deployments over docker container deployments.

Java Environment

Java Virtual Machine is one of the most used execution environments to run software from a wide range of programming languages (Java, Kotlin, Clojure, Scala, etc.).

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

Programs written in Java have a reputation for being slower and requiring more memory than those written in C++, but *Just-in-Time* (JIT) compilation, embedded in the JVM, delivers a boost of performance by opportunistically compiling *bytecode* to machine code at runtime. The JIT combines two compilers, C1 and C2 (also known as Client and Server VM), which are triggered based on the activity of the hosted application. Additionally, Java uses an automatic *garbage collector* (GC) to manage memory in the object lifecycle and recover the memory once objects are no longer in use. Each JVM usually includes multiple GC, each designed to satisfy different requirements. By default, HOTSPOT uses both C1 and C2 as tiered compilers,[6] and the *Garbage-First* (G1) GC with a maximum number of GC threads limited by available CPU resources and heap size, whose initial size is 1/64 of physical memory and maximum size may reach 1/4 of physical memory.[7]

Some studies investigated the impact that a JVM and its configuration could have on the performance and energy consumption of software. Oi [104] conducted a performance analysis and comparison between two JVMs: HOTSPOT and J9. They discussed in their results that the relative performance of a JVM depends on the workload. In their experiments, the performance of HOTSPOT ranged from 44% to 289% of J9, while its dynamic power consumption varied from 2.7W to 7.2W with the SPECjvm2008 benchmarks.

HotSpot and J9 were also compared in other studies. Chiba et al. [29] evaluated the effect that those 2 JVM platforms could have on the performance of a combination of big data query engines (SPARK and TEZ), using TPC-DS benchmark. They reported on a 3-fold drawback that one JVM can exhibit compared to the other.

On another note, Lafond and Lilius [74] attempted to assign a constant overhead to the JVM usage and assess the energy cost of atomic *bytecode* instructions in order to classify the most and least energy consuming *bytecode* instructions. The authors used a KVM environnement and ARMulator to emulate the JVM. One of the reported results is the constant distribution in energy consumption between the processor and memory over the execution, along with the high energy consumption of the memory access (70%).

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

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

A similar idea was used to design a model for JVM-based software energy consumption, using a *bytecode* level model [26]. The authors described their tool OPACITOR — that sums the energy consumption of each Java *bytecode* instruction — as being deterministic, accurate, and robust to the surrounding noise. However, they disabled the JIT and GC in their experiments to maintain the deterministic behavior of their tool, which does not reflect a real software execution given all the optimizations that the JVM triggers to optimize the software through JIT.

2.3.2 At Code Level

This might be the optimization level that has been the focus of most studies and contributions towards reducing SEC. The opportunities to reduce SEC cover two main axes.

- The first one is about the energy efficiency of the produced software, where the aim is to reduce the energy consumption of the final product with selecting the most adequate set of programming languages, tools, libraries, etc. It also includes all the efforts and optimization that developers can apply on source code to enhance the energy efficiency of the software during the operational phase;
- The second axis focuses on reducing the energy consumption of the coding phase itself. It includes all the methods, techniques or tools that can be used to reduce the energy consumption of the coding phase by limiting the wastes of resources induced by development and maintenance tasks. An example of such method is choosing green development tools and procedures.

Reducing SEC at code level includes: *i)* Identifying the best set/combination of programming languages that will allow producing the least consuming software, *ii)* Defining and measure the key performance indicators (KPIs) to evaluate and improve the source code energy efficiency, *iii)* Evaluating the development tools (IDE, libraries, APIs) to reduce the energy consumption of both the development phase and the produced software, *iv)* Investigating green coding best practices and guidelines, *v)* Adapting the source code to the execution environment capabilities for optimal performances and energy efficiency.

Many works have been conducted in order to reduce the energy consumption at code level. For the energy efficiency of the coding phase itself, some works have evaluated the energy consumption of some development tools. Kumar et al. [72] for example present in their paper an early experience where they compared the energy consumption of some Java development tools. They claimed across experiments that Intellij consumes less energy for JVM calls compared to Eclipse and Netbeans. In another example, Strubell et al. [136] compared the training phase power consumption and carbon footprint of multiple deep neural network models for natural language processing. The study showed that training an inconvenient model can cause a huge loss in power consumption and CO2 emission.

For the energy efficiency of software products, numerous works tried to provide developers with knowledge and guidelines on how to reduce the energy footprint of their software. First, some works aimed at helping developers choose an energy-efficient combination of programming languages for their software development.

Pereira et al. [118] conducted a comparison study of the most used programming languages in terms of energy consumption. The work also delivers suggestions on how to combine some of these languages to optimize the code quality considering the execution time, the memory usage and the consumed energy. Some of the results of this study is the bad energy efficiency that the interpreted languages—such as python—exhibit, compared to the compiled ones such as C or Rust. The paper also suggests combinations of languages that developers could use together to achieve a better energy efficiency, execution time and memory usage.

Other works investigated software energy consumption efficiency through source code changes and optimizations. For example, some papers [42, 122] studied the effect of Java collections on energy consumption, with depending on the collection size and/or the most executed tasks on the collection (insertion, removal, search). They provided some insights on the energy efficiency of some collections for multiple scenarios. For example, Hasan et al. [52] compared the energy consumption of several Java data structures, analyzing the bytecode using the Wala framework[8] and assessing the evolution of the energy consumption in different scenarios (insertion at the beginning, iteration, etc.). They also used some automated replacement of `LinkedList` and `ArrayList` to simulate best- and worst-case energy consumption scenarios on real production applications. Their study showed that using inappropriate collection can cause up to 300% of energy consumption inefficiency.

SEEDS and SEEDS-API is a fully automated framework, presented by Manotas et al. [90] to analyze source code (at bytecode level for Java) and auto-tunes apps to reduce their energy consumption, with a focus on Java collection tuning. The authors reported on up to 17% improvement using their framework. For I/O libraries' energy consumption, Rocha et al. [127] conducted a comparative study of some I/O methods, mostly I/O classes that inherit from `java.io.In(Out)putStream` and `java.io.Reader(Writer)`. This preliminary study offers some interesting insights on the behavior of some of the most common native Java I/O methods. Nevertheless, the reported work lacks some of the most used I/O methods, and does not deliver guidelines to reduce the energy consumption across multiple I/O scenarios and use cases.

In another work, Pereira et al. [117] presented SPELL, the energy leaks detector tool. The tool uses JRAPL [7, 122] to measure the energy consumption and instrument the source code. It detects energy-inefficient code fragments using a statistical spectrum-based energy red spots localization. The authors describe it as being language and context independent.

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

Jagroep et al. [62] discussed how energy profiling can be applied and provided an in-depth analysis of SEC across releases. To do so, the authors performed 2 empirical studies: an experiment to compare the EC of a commercial software Document Generator product across numerous releases, then a semi-structured interview with stakeholders. Similarly to Mancebo et al. [87], Sahin et al. showcased in their study how the energy consumption can increase between software versions due to the insertion of new features. In their study it was the introduction of an encryption module that caused a noticeable increase in energy consumption between two versions of the Document Generator software, measured using both WattsUp (hardware) and Joulemeter (software) tools.

Other works investigated the energy consumption of Java primitive types, operations on strings, usage of exceptions, loops, and arrays [72, 84]. For example, Kumar et al. [72] measured the energy consumption of code snippets and micro-benchmarks and presented some observations, such as string concatenation consuming less than `StringBuilder` and `StringBuffer`, static variables consume 60% more energy compared to instance variables, etc.

Mobile applications also witnessed numerous studies to reduce the energy consumption and save battery life. Corral et al. [33] implemented software benchmarks issued from the Computer Language Benchmarks Game (CLBG) in Java (Android App) and C (native and through JNI) They reported on equivalent energy consumption for small Java and C jobs. However, deporting some Android Java code and running it as a C code on a server can be more energy efficient for large jobs (only for battery life as the network/back-end energy consumption has not been considered).

Banerjee and Roychoudhury presented a lightweight technique that encodes the optimal usage of energy-intensive hardware resources in an application [17]. The authors claim that it can assist in energy-aware app development. Their work highlighted 4 main guidelines to save energy on a mobile application: *i)* Resources must be acquired as late as possible and released as early as possible, *ii)* Resources acquisition should not be nested, *iii)* QoS can be traded-off to improve energy-efficiency if the context permits, *iv)* All resources acquired during the execution of the app must be released before the app exits. Experiments on open-source Android applications demonstrated up to 29% of energy savings.

In a similar context, Rodriguez [128] presented some early experiments on different micro-benchmarks and discussed many coding aspects with a focus on implementation techniques, such as how to iterate on a matrix, avoid operations with immutable data types, evaluating strings, or the use of smaller numeric data types to save battery life.

Code Refactoring

Code refactoring encompasses the non-functional changes that developers apply on the source code to improve various quality aspects, such as readability, maintainability, etc. Achieving software energy efficiency through refactorings has been massively studied, especially for mobile applications [5, 48, 57, 81, 113].

For instance, EARMO proposes a multi-objective refactoring approach to automatically improve the architecture of mobile applications [99]. The authors conducted an empirical study to measure the negative impact of 8 anti-patterns on 20 open-source applications. They then used a multi-objective search-based approach, called EARMO, to correct up-to 84% of the anti-patterns on the tested applications and increase the battery lifespan by up to 29 minutes. However, their statistical analyses with a significance level of 5% only showed that half of the rules can impact energy efficiency in some cases. Moreover, the CPU/chip energy variation has not been taken into account for the significance level of the comparisons.

In the same context, Cruz et al. [35] present the LEAFACTOR tool to reduce the energy consumption of Android applications. The tool automatically changes the source code of the application by applying a set of patterns known to be energy efficient for mobile applications (such as `DrawAllocation`, `ObsoleteLayoutParam`). About 222 changes were submitted to the original projects repositories, resulting in a total of 59 Pull requests for 140 tested applications issued from F-droid.[9] At least 16 Pull requests have been successfully merged.

Anwar et al. [11] also gave concrete examples on how to save some battery time through refactoring. They achieved a maximum of 10% of energy savings by refactoring the DuplicatedCode and TypeChecking code smells. They also reported on a strong correlation between the impact of a refactoring on the execution time and the energy consumption.

Furthermore, Cruz and Abreu [36] studied the effect of 8 of the best performance-based practices on the energy efficiency of 6 Android applications. The results of the experiments showed that some patterns, such as ViewHolder, DrawAllocation, WakeLock, ObseleteLayoutParam need to be taken into account for a better design of energy-efficient applications, with a reported impact of 4.5% for the Writeily-Pro application.

Finally, Moreira et al. [100] analyzed a set of 16 tools from the literature that reduce the energy consumption of software through refactoring, with a list of 11 code smells. They discussed the weak liveness of the available tools (requires manual tasks to be tuned and triggered). The paper summarized an average energy impact of 1.9% and a maximum of 4.5% for the 11 refactorings, but did not discuss the relevance of these impacts neither the eventual causes nor measurement errors, especially that 30% of the mentioned refactorings have less than 1% registered impact.

Most of the works reported on the presence of an impact of refactoring rules on energy consumption. This can be substantial or relatively small depending on the application. Yet, most of the covered patterns are related to screen/sensors usage that are very specific to mobile applications and cannot be generalized to other systems/environments.

[9]https://www.f-droid.org/

Thus, other works investigated the impact of refactoring on server-side and desktop applications. Pinto et al. discuss 12 contributions taken from the state of the art on the refactoring that can be applied to improve software energy efficiency [121]. This literature review was conducted on the papers that were published in 8 of the top software engineering conferences prior to 2015. It summarizes some interesting information and practices relating to CPU offloading, HTTP requests, I/O operations, DVFS techniques, etc. Sahin et al. [130] also studied the impact of 6 refactoring rules on a total of 197 selections found in 9 Java applications. Their results showed that the impact of applying the refactoring could be statically significant, but is not very consistent across the software and platform versions. They suggested that knowledge on the energy consumption impact of refactoring rules could be integrated within IDEs to help developers build less energy-bleeding software.

In a more detailed study, the impact of only one refactoring rule "*inline method*" has been investigated on 3 Java applications [147]. It reported that the impact on the execution time and energy consumption that was expected to be positive, was not always true. This means that the *inline method* refactoring is not always energy efficient.

Rather than looking for green refactoring rules reducing software energy consumption, some practitioners chose to conduct wider studies that apply on a much larger set of refactorings to capture a subset of "green" rules. This is exactly what Jae-Jin Park et al. [60] pursued. They prepared C++ micro-benchmarks of 63 refactoring techniques/design patterns suggested by Martin Fowler [1], then ran experiments and isolated a set of green refactoring rules based on the micro-benchmarks for C++. However, the conclusion cannot even be generalized on C++ applications, as they were built on specific tests that were executed on specific micro-benchmarks.

Automatic and search-based refactoring is also an interesting topic that has been covered by many papers to reduce software energy consumption [107]. For example, Moghadam and Ó Cinnéide [98] presented Code-Imp, which is a Hill Climbing search-based tool that enhances the application quality at field-level and class-level refactoring, where the fitness function can be extended for energy consumption purposes.

2.3.3 Summary

This chapter gives an overview on the state of the art of software energy consumption. We thus reviewed a set of hardware and software tools that allow measuring and assessing SEC. This measurement phase is subject to instabilities and variations due to multiple factors. These variations should be carefully taken into account to conduct robust and accurate measurements.

Some of the studies aimed at highlighting developers knowledge and sensitivity to the topic, so SEC could gain more importance and consideration.

Reducing software energy consumption is a non-trivial topic that has been investigated in several studies. Many studies investigated ways to reduce SEC through optimizations on the source code of the software and/or the execution environment. A set of insights and guidelines can already be extracted from these studies in order to produce and deploy less consuming software.

Chapter 3

Understanding the Hurdles and Requirements to Optimize Energy Consumption

In the precedent chapter, we reviewed some studies aiming at reducing software energy consumption. In particular, developers awareness and knowledge about SEC considerations. This chapter describes the opening work and contribution. Its purpose is to investigate and discuss the understanding of green software design among developers and within the company. We believe it is the first element to apprehend in order to define the perimeter of developers needs and requirements. This qualitative study provides implications for developers, decision makers, tools creators and researchers to promote green software design. The contribution covers three main questions: 1) developers understanding and knowledge about SEC and GSD, 2) the constraints and tooling lacks that prevent a good support of GSD, 3) and, how to sensitize and promote GSD among developers. We start this chapter with an overview in Section 3.1 that motivates and situates the work that has been achieved. Then, we formalize our methodology in section 3.2. Section 3.3 analyses and discusses the observations and findings behind the interview answers. Section 3.4 reports on the implications of our findings.

3.1 Overview

The last decade witnessed several attempts to consider green software design as a core development concern to improve the energy efficiency of software systems at large [12, 17, 72, 89, 101]. However, despite previous studies that have contributed to establish guidelines and tools to analyze and reduce the energy consumption [10, 32, 50, 67, 69, 99, 123], these contributions fail to be adopted by practitioners till date [62, 114].

Concretely, both quantitative and qualitative studies [91, 114, 119] previously surveyed developers to establish assumptions about developers' knowledge of green software design. These studies highlight that developers might be aware of software energy consumption problems, but have a very limited knowledge on how to reduce the energy footprint of their software product. Understanding the hurdles and requirements to optimize energy consumption. For example, Pinto et al. [119] mentioned collecting "vague" answers from developers when asked about how to deal with software energy consumption. Pang et al. [114] reported that, among 100 developers, a small portion are aware of the primary sources of software energy consumption. Only 10% of the participants try to measure the energy consumption of their software project, while less than 20% take energy into account in the first place. Moreover, the empirical study of Manotas et al. [91] reported that energy requirements are often more desires than specific targets. They highlight that developers believe they miss accurate intuitions about the energy usage of their code, and that energy concerns are largely ignored during maintenance.

However, to the best of our knowledge, none of these studies discuss

RQ 1: The hurdles that prevent the broader adoption of green software design? and

RQ 2: Developers' requirements in terms of tooling in an industrial context.

But, we actually believe that both aspects are critical issues to consider when aiming to reach an adoption of such tools and methods among developers in order to promote green software design.

This chapter summarizes our qualitative investigation on software energy consumption considerations among experienced developers at Orange France. Concretely, we conducted interviews with 10 senior/expert developers with the ambition to cover developers' opinions, problems, and requirements to promote the green software design in an industrial context. The key contributions of this chapter can, therefore, be summarized as:

1. Providing a detailed understanding of the interviewed developers' awareness and knowledge about green software design,
2. Identifying the main constraints and challenges that developers encounter in their daily development,
3. Building specifications for the tooling that suits developers expectations and experiences,
4. Investigating the best ways to keep developers aware of software energy consumption and promote it within a company,
5. Identifying the exact role and responsibilities of the company to promote green software design,

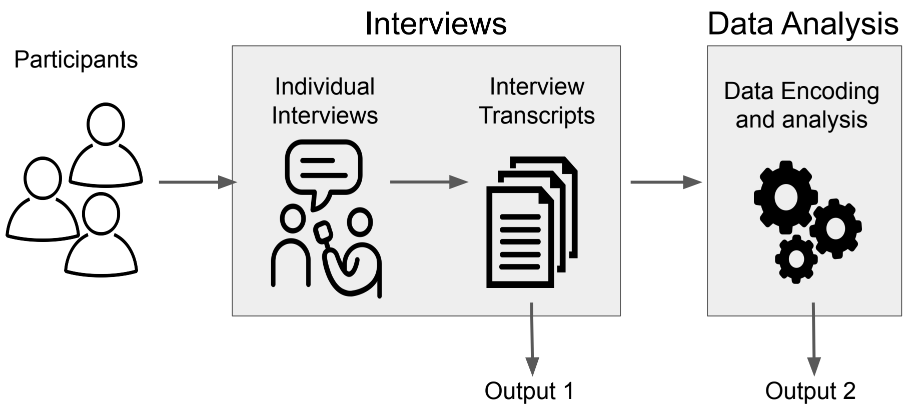


Figure 3.1: The qualitative study methodology

3.2 Methodology

In order to achieve our objective, which is to conduct an in-depth qualitative study that encompasses developers tooling requirements and awareness, we adopted a qualitative research approach [34, 116], using straussian grounded theory concepts and components, such as: *coding*, *memoing* and *theoretical saturation* [135]. Despite being complex and time consuming [18], this approach has been widely adopted by similar studies and has proved its effectiveness [45, 51].

Figure 3.1 depicts a high-level summary of our qualitative study methodology. Our methodology starts with an interview phase detailed in Section 3.2.1, followed by the data analysis phase explained in Section 3.2.3. We discussed our methodology with a qualitative study expert at Orange to ensure the good usage of the theoretical aspects and other criteria, such as data confidentiality and integrity.

3.2.1 Interviews

Interviews are the first step and the main data source for our qualitative study. In this study, we wanted to cover 3 main research questions. The first research question is the awareness and knowledge of developers with regards to the software energy consumption. Even if this was the focus of many papers, like [114, 119], it is still very important to investigate participant's opinions about software energy consumption, as it helps to better analyze his/her others answers, depending on what he/she thinks of the problem and how important it is. The second research question aims to investigate about the hurdles and constraints that prevent a better consideration of software energy consumption, but also to push the developer to define and describe the tools that will suit his/her experience, to promote the consideration of software energy consumption in his/her daily development. The purpose of the third research question is to identify the best ways and means to keep developers aware of software energy consumption, but also to zoom into the responsibility of developers on one hand, and the decision-makers—or the company—on the other hand.

During the interviews, we wanted to give our participants the freedom to express and explain their ideas and opinions so we can gather more feedback. Thus, we went for semi-structured interviews. The following sections provide more details about participant's selection, interviews conducting protocol, and the questions we asked.

Participants

The main criterion for the participants' selection is their experience in software development. Experienced developers in each technology had more time to cover the details, strengths, and limitations of the technology they are using. A junior developer in a specific technology or programming language may not have enough time or experience to cover all the basics, best practices and go deep into the technical characteristics, and is less likely to include energy considerations in his/her coding routines. By experience, we do not mean the professional experience, but a decent amount of time the developer has spent on a technology/programming language, to have enough knowledge to understand and be able to criticize this technology. Thus, our participants have experience of at least 15 years and have worked on both small/short and long projects. Moreover, our selected participants are volunteers who have expressed a big interest in our interview invitation to cover developer's understanding and requirements regarding software energy consumption considerations and are more involved in green software design activities in a major European telecommunication company of more than 100000 employees. The rationale behind choosing participants from the same company [47] is to assess the role of the company in the practice of developers. However, our study focuses on how to promote green software design within a company and expose a detailed case study but does not ambition to create a model that could be automatically generalized to companies of different sizes, activity sectors, or policies.

We broadcasted emails through the internal senior and experts mailing lists, which regroups experienced developers in a wide range of technologies and projects. Then, we selected our participants so we can cover numerous technologies, including mobile, web, etc. We were also careful to select both developers working on long projects and others working on more frequent short projects. Such a diverse selection process in a qualitative research method has been described in Patton, Michael Quinn's book [116]. Table 3.1 summarizes some useful data to understand the profile of every participant including his/her experience, main used technologies/programming languages, and projects type. We use the letter "L" for long/big projects, and "S" for small/short projects.

Participant	Technology	Experience	Projects
P1	Mobile Python	10 years 10 years	L
P2	Mobile Java	10 years 20 years	S and L
P3	Java	20 years	S
P4	C Java	10 years 10 years	S and L
P5	Java Golang	10 years 5 years	L
P6	Web	20 years	S
P7	Web	20 years	L
P8	Java Web	12 years 7 years	L
P9	Java	11 years	S
P10	Java Golang	12 years 7 years	S and L

Table 3.1: Summary of participants

Instead of rigidly fixing the number of participants, we kept on conducting our interviews until reaching a level of saturation on the collected data [131]. After 10 interviews, we noticed a convergence of the collected data and thoughts [145], even considering the difference of technologies mastered by our participants and the types of projects they usually work on. Moreover, 10 is a decent number of participants that is close to the studied population by other similar works [22, 51, 133, 138].

For privacy and confidentiality purposes, we omit the usage of our participant's names and every other sensitive information, such as teams or project names, and we rather use code names ranging from P1 to P10.

Protocol

The interviews were conducted in 3 steps. The first step is a narrative part where we describe the purpose of our study, what the interview is about, and how it would happen. It also includes the confidentiality agreement with the participant and some indications of the interview process.

The second step is the semi-structured interview, starting with questions about participant's profile, which cover: participant's studies, the type and examples of projects he/she worked on. Then, we continue with the interview questions that focus on the 3 research questions introduced earlier and listed in Section 3.2.1. Finally, we conclude the interview with a post-questions step, where we answer participant's questions and share some information and references if she/he is more interested in software energy consumption.

Our protocol was checked and assessed by a qualitative studies expert from the company, before being tested on two developers—whom results are not reported— to apply some adjustments on the questions and the interview scenario on one side, and have a better duration estimation to inform every candidate of the average time before every interview on the other side.

To make the interview very fluid and capture every information, we recorded (with the agreement of the participant) the second step of the interview to apply post-in-depth analyses. We also prepared a quick summary sheet that allowed us to note the key answers for each question, along with participants' key thoughts and opinions. This mainly helped us to quickly detect the data saturation, as suggested by the qualitative studies expert, before the detailed analysis phase that confirmed it.

Three of the interviews were held face-to-face. The others were conducted via a call due to the distance between the interviewer and the interviewee sites. Also, all the interviews were done in the native french tongue of the participants to avoid any misunderstanding or expression difficulties due to the language. The mean duration time of the interviews is 39 minutes and 36 seconds, with a minimum duration of 28 minutes and 13 seconds, and a maximum duration of 54 minutes and 09 seconds.

Questions

Using semi-structured interviews was very helpful in our case. It allows identifying the main questions defining the purposes of our investigation. It was supported by follow-up questions to adapt to the participant's answers and let explore more details and directions in their answers. The main questions have been pre-defined and structured before the interview so the process goes faster, and to keep track of our baseline questions and concerns. We gave special attention to the formulation while preparing the questions. We wanted them to be open so we do not get a *"yes"* or *"no"* answer, but also to go deep in every participant's answer with the follow-up questions, as long as we maintain the theme of the main question. The main questions we asked are the following:

1. What do you know about software energy consumption and green software design?
2. What importance do you give to software energy consumption?
3. What are the software energy consumption considerations that you take in your developments?
4. What do you think are the constraints and hurdles to a better software energy consumption consideration?
5. How ready are you to change your usual programming language, technology, library, for better software energy consumption?
6. How do you describe perfect tooling that suits your coding requirements for green software design—you can go deep into technical details?
7. How do you think we should inform about energy software consumption for better awareness?

8. Do you think that getting a better software energy consumption consideration is the responsibility of the developer or the company? How?
9. How can green software design be used as a marketing argument?

The questions (1) to (3) cover participant's knowledge and awareness of software energy consumption. The question (3), in particular, investigates any experience with methods, tips or tools that the participant has used for green software design purposes. Questions (4) to (6) aim at discovering the constraints that developers encounter and their tooling requirements, for better software energy consumption considerations. The purpose of the last three questions is to learn how to improve awareness. For that, we look for the best communication channels that developers would react to, to promote their consideration of the green software design. The last question is a more open one that summarizes the participant's belief and gives him/her more freedom to discuss some points that we might have missed during the interview.

Depending on the participant's answers, we ask some follow-up questions which are guided by the theme of the main question and the content of the answer. One example of a typical follow-up question we had to ask quite often along with the question (5) is: Have you questioned the quality of a tool/method/technology you have been using for a long time during your experience? How was that?

3.2.2 Transcription

After each interview comes the transcription phase and we opted for a denaturalism approach to transcribe our records. This method has been used in similar works [51], and allows putting a focus on the interview content while being lighter, but as complete and trustful as other methods, like Verbatim [105].

The transcription was made in the same language of the interview, but we translated some parts in English to quote participant's opinions in Section 3.3.

Some of the participants agreed only on sharing the results of the study, but not the raw data (recordings and transcripts). Nevertheless, we worked on preserving the participants' privacy, by omitting project names for example.

3.2.3 Analysis

We based our data analysis on the Straussian grounded theory coding procedure [135, 148]. First, we started with the *open coding* phase, where we read our transcripts several times and tried to summarize every chunk of data into a label, based on the meaning interpretation of the text. These labels are called *"open codes"*. Next, we used *axial coding* to identify the connections among the previously extracted open codes. Then, we used *selective coding* to figure out the core ideas, which cover all the data we collected. Finally, we read the transcripts again and selected any data that relates to the core ideas so the content segments of the transcripts will be all assigned to a core idea.

The analysis has been independently conducted by two different persons to increase the accuracy and hinder the subjective interpretation overhead. The results were then compared and discussed for a consensual decision.

3.3 Observations and Findings

Table 3.2 summarizes the key results of our study, with the core ideas that also match our main objectives. The check-mark (✓) in each cell indicates a positive response from the participant regarding every idea that the core idea encompasses. This section discusses our observations, each subsection covering a reported idea. Every single idea of Table 3.2 is then discussed in a dedicated paragraph. Ideas that express close meanings and purposes are grouped within the same category. We provide a discussion at the end of every category to summarize the observations and findings of the detailed ideas and to add our thoughts and recommendations.

Core ideas	Ideas	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Developers awareness & knowledge about SEC	I already know about SEC	✓	✓				✓	✓	✓		✓
	I already considered SEC in my projects		✓				✓	✓	✓		
	SEC is an important subject to consider		✓	✓		✓	✓	✓		✓	✓
	SEC should be a high priority			✓		✓	✓	✓			
	SEC might cause conflicts with other coding metrics/aspects		✓						✓		
Constraints & tooling problems	No time to think about SEC	✓							✓		
	No tools			✓				✓	✓	✓	
	The main problem is not at the developer level		✓	✓					✓	✓	✓
	Ignorance		✓				✓	✓		✓	✓
	Enhancing performance often enhances the SEC		✓		✓	✓					
	Need for a SEC score/KPI	✓		✓	✓	✓	✓	✓	✓	✓	✓
	Include SEC among tests/CI	✓		✓	✓	✓		✓	✓		✓
	Static code analysis	✓				✓			✓		
	Simple tool with simple outputs	✓		✓	✓		✓	✓		✓	✓
	In Favor of Moving to Other Technologies / Tools			✓		✓	✓	✓			✓
Promoting SEC	The company has most of the responsibility compared to devs	✓	✓	✓	✓	✓		✓	✓	✓	✓
	The company should put objectives around SEC	✓			✓	✓		✓	✓	✓	
	The communication about SEC should be improved	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Need for training				✓						✓
	Simple presentations are effective	✓		✓	✓	✓	✓	✓		✓	
	Let's put green labels on software	✓		✓			✓	✓		✓	

Table 3.2: Summary of our interview analysis.

3.3.1 Developers Awareness and Knowledge About Software Energy Consumption

Developers Knowledge About SEC

The level of knowledge of the participants on software energy consumption is disparate. Some of the interviewees reported having some knowledge about green software design, or even considered it in their projects, while others reported complete ignorance on the topic.

I already know about SEC Software energy consumption is a relatively recent subject that people may or may not know about. For some of our participants, they heard about it on many occasions lately. *"I have attended several talks on software energy consumption problems, and have seen several things"* reported P1. Some of the participants reported never hearing about software energy consumption (P3, P4, P5) with answers like *"nothing"* or *"absolutely no idea"*. Others have shown a very recent interest in the subject using expressions, such as *"I became interested in software energy consumption a couple of months ago"* (P6, P7, P10).

However, even among developers who answered the question affirmatively, some had a hard time explaining what do they know about software energy consumption and gave global optimization examples, which are not specifically related to energy consumption. *"I know that reducing code size is good for energy consumption"* said P6, while P7 reported *"I have some ideas about web applications, such as reducing the size of data we send to the user"*.

I already considered SEC in my projects Among the participants who reported knowing about SEC problems, 4 claimed they have already applied software energy consumption related practices in their projects. P2, for instance, witnessed *"I try, but it is not easy [. . .] we avoid to do useless animations [. . .] limit the access to servers [. . .] we keep an eye on the battery so our software does not drain it too fast"*. P6, P7, and P8 reported on attempts to reduce the software energy consumption through enhancing the performance, *"We try to cache data and limit the transfers [. . .] the main focus is the performance but also the energy by chain effect"* said P6. For P7, he/she is trying to get more involved in considering green software design in his/her projects. *"I think making sure the mobile application works on old phones is a good example"* he/she shared, confirmed by P2.

Discussion Developers confront numerous kinds of information from multiple sources. Such sources do not always constitute a valid/correct set of knowledge. In the case of our study, some of the participants heard about software energy consumption, but could not provide a correct formulation of their knowledge without diverging from the energy consumption.

We argue that developers awareness can be classified into 4 different levels: 1. not knowing about software energy consumption, 2. having wrong/incomplete knowledge about the issue, 3. stacking theoretical knowledge with no application, and 4. knowing and applying SEC considerations. We evaluate most of our participants to be at the 2nd stage, while proper communication and training programs should be established to help developers reaching the 4th stage.

SEC Importance Among Developers

Our study shows that not all the participants give the same importance to green software design. The participants reported different levels of awareness, from being not important to be one of the highest priorities in software development.

Importance of SEC While most participants think that software energy consumption is an important issue, some think that little attention is generally given to it: *"Pretty low importance"*, *"None, absolutely none"* and *"The most important thing is to deliver the service to the consumer"* (P1, P4 and P8, respectively). These answers are more related to their professional environment and work, rather than their own opinion and personal considerations. P8 added *"If we are talking at the societal level, then energy consumption is important, but it is not the case for what we are producing at work"*. The other participants feel like the matter is quite important and should be more considered at work. P5 and P10 even shared that the professional environment can support the movement even better *"As we trend to sobriety"* said P5, pointing at the company's objectives towards sobriety and greenness. *"It is part of the current challenges"* reported P10 in the same context, referring to the newly announced environmental objectives of the company.

Priority of SEC Being important is a thing, but being a priority is different. Among the participants who reported the importance of software energy consumption, P2, P9, and P10 think that it is important but, at the time of the interview, not a priority in the company. *"It is one of the main challenges but it will be utopian to think it is a priority"* argued P10. *"From a company point of view, it is not the priority"* added P2. This shows a different understanding of company strategy and priorities. P4 symbolizes it, by answering *"Zero, but I might change my mind if I get persuaded that it is not the case"* when asked about his perception of the current priority level of green software design within the company. On the other hand P2, P6 and P7 see it as a priority that the company has pushed in the last decade. *"The green aspect is ubiquitous"* said P6.

Discussion While discussing the value and the significance of software energy consumption, we got more evidence about the problem of communication. We can see that not all developers were on the same level of awareness, knowledge, and even trust towards the company [114]. While all of our participants claim to be in favor of green considerations in their personal lives, some developers do not think the matter of software energy consumption is important in the current software development processes at the company. Some participants think it should be a high priority in daily coding tasks, while others give it the same importance as several other code-related aspects (security, maintenance, etc.).

This also highlights the differences in the trust in the company. While some participants are in favor of the strategy of the company, others think that it should be improved. For this matter, we think that the company should conduct a large transparent internal campaign, to precisely identify the objectives and requirements towards the employees, and clarify any misleading ideas, thus ensure the same comprehension inside the company.

SEC Might Conflict with Other Considerations

P2 and P8 reported that software energy consumption considerations might cause potential conflicts with other software metrics and cause a lower rate of maintainability, security, scalability, etc. On this matter, P2 reported *"today, if I have to choose one thing over the other, I would choose code maintainability over a lower energy consumption"*. P8 has also a similar opinion *"I would only consider software energy consumption in a third position, once we ensured that the service is well optimized, and the security is guaranteed, which is one of our biggest concerns"*. Some other participants as P6 did not evoke conflict and talked about including the new metric among the other existing metrics.

Discussion While most of our interviewees are confident about software energy consumption integration within the daily coding considerations, some legitimate questions arise on how this integration would happen. We are not safe from conflicts that might occur. Choosing a less consuming, but harder-to-maintain programming language, or substituting some consuming security methods, are examples of those potential conflicts. Hence, a need for a well-designed strategy is required within projects. This strategy could include a set of objectives and validation steps per project, that derives for a more global set of objectives, to ensure taking the same choices and achieving the same purposes, as defined in the global strategy.

3.3.2 Constraints and Tooling Problems

Work Constraints

Some challenges have been raised by our participants to express the difficulties regarding software energy consumption considerations.

No time to think about SEC *"No time"*, is the answer that two of our participants provided. P1 reported *"In our projects, we do not have any free time"*, meaning that with all the considerations that a developer has to take into account in a project, he/she cannot afford any extra time to deal with software energy consumption issues, at least not if no time was specifically allocated for that purpose. P8 mentioned the same issue, highlighting the potential conflict between the extra development time required when using less energy-intensive technologies, and the allocated time: *"The time factor is never negligible in our projects, and the least consuming technologies tend to require more time"*.

No tools P3, P7, P8 and *P9* miss the appropriate tools to dive further into software energy consumption problems. *"I never heard of such tools or what they do"* is what they reported. This lack of tooling, and the related lack of feedback on actual energy consumption, hinder the analysis of the root causes of SEC, and thus the potential actions they could do to recover from bad energy practices. For example, P7 reported: *"we do not have any feedback or indicators"*. *"The main problem for me is the lack of tools, we do not have automatic tools for green code quality"* shared also P9 to express some frustration due to the absence of tools for green software design.

The main problem is not at developer level For some of our participants, the main problem is not at the developer level. In fact, for P8, P9 and P10 the problem is at decision-making level. The developers being only the execution unit, they do not have much impact once the decisions have been made: *"we do not have full decision power"* said P2. P8 reported *"I am not sure if this is the crux of the problem, but we should be able to provide the developers with the proper tools first so they can achieve this purpose"*. This points to a lack of tools, but more importantly, to the role of the whole chain to organize and define the work conditions, towards a green software design.

Ignorance By "ignorance", the participants refer to the lack of knowledge, but also the lack of awareness about software energy consumption. P2, for example, reported a problem when designing mobile application interfaces. *"I have some requests to put animations all over the screen, which does consume a lot of energy and does not improve the user experience by much"* he/she said. This ignorance is also illustrated among some developers, *"Ignorance is the first reason, I did not know a thing myself, developers do not know what they can achieve and the impact they could have"* claimed P7. P9 and P6 share the same opinion, *"People think resources are endless"* reported P6. For P10, this ignorance problem might even dissuade good-wills who want to change *"The problem is that we do not all share the same green culture at the company, if the team we work with is not on the same page, nothing will happen. We need to be all on the same level"*.

Discussion The participants we interviewed expressed a list of constraints and hurdles that prevent software energy consumption considerations. Among these constraints, we can identify the lack of time, tools, and awareness. Indeed, the developers would feel much more comfortable about green software design if they had dedicated tools that support that activity, and a decent awareness so they can set software energy consumption related objectives at the beginning of every project for example. A dedicated time is also required. By this, we mean allocating a specific period so the developers would be able to set up the green software design environment and checks, but also allocating the time that developers need to get acquainted with the potentially new aspect of energy in software development.

The inclusion of the decision-makers when talking about software energy design has also been a matter of discussion in some interviews. Some project's key decisions may have a big impact on the energy efficiency of the final product. Choosing the proper technologies and allocating the proper time can be good examples of that, where developers do not have much choice and just try to deal with the constraints and deadlines.

Enhancing Performance Often Enhances SEC

While most of the participants diverged into performance metrics when asked about energy, some of them reported a correlation between the performance of software and its energy consumption. *"There is certainly a direct link between performance and energy consumption, for mobile applications. The more we ask the phone to do the quicker it drains the battery"* claimed P2, and *"Theoretically performance and energy consumption are not the same things, but in practice, they are"* reported P4. For other developers, like P3 and P9, the causality relationship is not that evident: *"Sometimes we spend a lot of extra energy while trying to enhance the performance"*. They even gave some examples like allocating more servers to go faster while requiring more energy.

Discussion Our participants have mainly experienced dealing with performance instead of energy consumption. Thus, they referred to performance several times instead and tried to replicate their knowledge on energy consumption. We think that "performance vs energy" is a mandatory topic that should be discussed when promoting software energy consumption, as all the developers should be able to distinguish the slight difference and knowingly take action that can favor performance over energy and *vice-versa* when there is a room for conflict.

Tooling Specifications

Gathering requirements for tools that would match developers' requirements is one of the priorities of this study. Fortunately, we identified some specifications that should help to design the next set of SEC optimization tools.

Need for a global score / KPI This has been the most requested and discussed specification. Almost all the participants mentioned the need for a global score or *Key Performance Indicator* (KPI) for the total software energy consumption evolution. *"We need to have indicators with a ranking system"*, *"consumption summary"*, *"We do not have any KPI"*, *"We need KPIs"* and *"track the evolution using simple KPI"* are claims from P1, P3, P6 and P7, respectively. P4 and P9 went a bit further and assimilated it to the consumption ranking that is used for household appliances, like washing machines: *"energy consumption classes like A, B, C, etc."*

Include SEC among tests/CI This also was a common proposition for many participants when asked about how they describe a tool that would measure and track the evolution of software energy consumption. They suggested for software energy consumption reports to be integrated within the existing system platforms that the developers are using, *"It would be a tool that is integrated with my CI chain to track the consumption evolution of my software"* reported P7. However, some developers do not think software energy consumption can be measured/tracked at the code level, as it dependents of the running environment that must be be modeled and simulated through testing and continuous integration, *"we could imagine the usage of micro-benchmarks to test the code quality on the same execution environment, which is not possible on the development station"* stated P5.

Static code analysis Some developers assimilated a part of green software design tooling to static code analysis tools, such as the Sonarqube tool, *"we could assimilate it to a Sonarqube to apply a first static audit and check some well-known bad practices"* mentioned P8. Others think that static code analysis is quite irrelevant for this purpose. P7, for example, does not believe in static code analysis as SEC is very dependent on the execution environment. *"We cannot have generic practices, we should specify the target, the platforms, etc."* said *P2*. *"It is also dependent on run-time"* reported P3. This shows that not all developers have the same trust towards static code analysis to diagnose software energy consumption issues.

Simple tools with simple outputs Participants also asked for simple tools with simple outputs, with the use of graphical interfaces to track the software global energy consumption's evolution, *"[...] with graphical output [...] that lets me notice the* 10% *energy difference"* reported P10. *"I should not need a Ph.D. to understand the outputs of the tool"* said P7 to illustrate his/her frustration with complex tools overloaded with too many details and no single score that defines the global status.

Discussion The participant's feedback about the tools was very rich and converged to the same main ideas, where usability seems to be the key concern. Developers expressed their requirements in terms of tooling, focusing on the simplicity of the outputs, which should include global KPIs/scores. When talking about energy consumption, the participants are very used to this kind of score in their daily life, with energy labels for household appliances, bulbs, house isolation, etc. Moreover, the same kind of scores is also routinely used in their daily development work with scores on CPU consumption, memory and disk usage, response time, etc. This global indicator should allow them to track the energy consumption evolution of their source code and would be an entry point to dive into the details about the consumption of a more specific code part, like a method or an algorithm. The static code analysis was a bigger question mark to some developers when speaking of its effectiveness. While it could be very beneficial to establish some rules about bad practices and common energy bugs [115] through linters for example, it still delivers limited feedback on the actual energy consumption at run-time. We argue that the discussed aspects should be taken into account by tools creators, where we could imagine a tool that applies static code analysis rules during the development phase (integrated with the IDE (Integrated development environment) for example), an energy profiler for code tuning, followed by a run-time energy tracking, integrated with CI/CD, and a dashboard for data visualization. The displayed information could be scores calculated from run-to-run evaluation, energy details/guidelines about the used technologies, etc.

In Favor of Moving to Other Technologies / Tools

Considering switching to another programming language, for example, is a legit question to ask when talking about a new aspect as software energy consumption. Some of the participants (P3, P5, P6, P7 and P10) reported no resistance for change. This openness is justified by the recurrent changes of technologies in their previous projects. *"I very often move from one technology to another, I have no problem with that"*, *"It would not even be a difficulty"* and *"Yes, it is a good thing"* reported P7, P6 and P10 to express their confidence in being able to move to other technologies for green software design purposes. Meanwhile, some developers worry more about this change: P9 explained that the choice of technologies is also related to the developer profile, and going for more energy-efficient but less used/famous technologies would be a bad decision for his/her career. For P1, software energy consumption is not important enough to justify such a delicate thing as changing the used technologies, *"It is hard to say, especially to re-develop the already existing software, choosing the programming language, for example, is very delicate, especially when software energy consumption is not a priority"*.

Discussion It is not surprising that some developers will express reluctance to change, as it is a delicate process that is influenced by both technical and social factors [76]. Especially if developers do not have enough knowledge of what they can do, how they can do it, and the reasons for such a change. It was however interesting to observe that some developers are in favor of such changes. This gives hope towards applying green software design. We argue that these changes should not be done in one shot, but in a couple of steps to make the adaptation easier for developers, especially after a good awareness and teaching campaign within the company.

3.3.3 Promoting SEC

The Role of the Company

As for the developers, the role of the company is important to establish green software design practices.

The company bear most of the responsibility compared to developers Except for P6 who does not believe in *"forcing"* the developers, but in *"If each person is aware, the collective will thrive"* instead. All other participants assigned the major responsibility to the company, to guide the application of green software design. According to P4, software energy consumption will never be a long term consideration if the company does not take the lead, *"We cannot have a hundred ways of doing, it has to be guided"* he/she added. *"You cannot stay in a corner and hope it will work, it has to be applied and guided through the whole chain"* argued P2. P3 compared the roles of the company and developers to a garbage selection process, where he/she assimilated developers role to *"putting the trash in the specific bins"* (small but important responsibility) and the company role to the recycling that is done afterward (big responsibility).

The company should include SEC objectives One of the company's roles that were fairly repeated during the interviews is setting objectives about software energy consumption and green software design. Many participants reported that setting objectives is a good way to promote SEC considerations. P5 argued *"we should have objectives around that, this would allow developers to see what they are doing and what they can achieve"*. Moreover, having objectives will give more credibility to the task, as it has been specified by products owners, *"having objectives will give more sense to green software design, and it will be one of the aspects that developers are going to be judged on by the end of the semester, as it has been specified by product owners"* reported P7. Those objectives could be related to the KPIs, *"identify some KPIs, and set some objectives around those KPIs"* reported P9.

Discussion Establishing green software design in the company is a matter that has to be supported by both decision-makers and developers. The company certainly has a backbone role in this process, starting with keeping the developers aware of green software design, providing them with the needed tools, and identifying global and project-related objectives. Setting objectives is very important for various reasons. First, it shows the dedication of the company towards green software design, which will transfer to the employees afterward. Then, defining a milestone would help developers to be more motivated to unlock a new achievement every semester/year regarding software energy consumption.

Communication Means

Now that we have seen the relative lack of awareness and knowledge many times across the previous discussions, we asked the developers what should be done to reach them.

The communication should be improved As pointed many times in multiple discussions in this chapter, the communication around software energy consumption and green software design should be improved. All our participants brought the communication problem at some point of the interview, by evoking the ignorance of the employees on these subjects (P2, P6, P7, P9, P10), or by describing the company as the guide to raise awareness about green software design (P1, P4, P5, P7, P8, P9).

The participants also gave some examples of the kind of communication they think they will be receptive to, like training programs, presentations, and conferences. P5, for example, proposed the usage of the company social network that is *"used by most of the employees"*.

Need for training Two of our participants reported a crucial need for training to learn the specificity of software energy consumption and how to manage it. P4 argued that training is very important if software energy consumption is a real concern today, *"A training over a couple of days to learn how to do things at developer level would be very welcome. We do training for other code aspects such as security, why not for software energy consumption"*. For P10, one solution is to train a group of developers to be experts in software energy consumption. These experts would then integrate project teams and would have a mission to help the other developers learn and apply green software design. *"Training all the developers would cost a lot of time and money"* added P10.

Presentations are very effective Many participants argued that presentations (informal presentations, presentations in conferences, etc) are effective to keep developers aware and inform them about software energy consumption and green software design. According to P3, *"I attended a presentation lately about green challenges and I found it very interesting, with real examples"*. The presentations should be instructive but simple *"including comparisons, pictures"* said P6. *"Ideally provide some tips with the related impacts, I have a book that enumerates* 115 *web best practices, that is huge, it should have* 15, *even* 15 *is still quite a lot"* added P7.

Discussion Communication is the keystone of every activity inside the company, and promoting green software design is no exception to the rule. The company should amplify and refine the communication around green software design, so it can be much deeper than just announcing a global plan. It has to allow the developer to act on the project he/she is working on, and to see his/her impact whenever possible. Our participants mainly mentioned two means of communication. First, the presentations should be very brief, very recurrent, and should include more examples and tips rather than flat knowledge. The presentations should be used mainly for awareness rather than knowledge transfer. Training is the second evoked way of communication, which is more knowledge-based, with more details than just tips, due to the specificity of training (much more time and fewer participants compared to presentations). We argue that a wise combination of these two means of communication would be a good start towards promoting green software design, as it allows having recurrent communication that involves most of the employees, to keep them informed about the objectives and the main guidelines. This can be achieved through presentations, meetings, and other formal or informal communication means. The second part of communication should provide the employees with the necessary knowledge to achieve that task efficiently, through training and workshops, etc.

Green Software Marketing

The purpose of the last question of the interview is to summarize every participant's belief in software energy consumption. The answers were split between two main ideas. While some interviewees (P1, P3, P6, P7, P9) did not hesitate to say that putting green labels on the produced software should create another selling argument, Other participants were very cautious about that label. P4, for example, reported *"I am worried about greenwashing, we have to be very careful about that"*. In the same context, P2 reported *"I am kind of worried that it will be used a little bit too easily, we have to be green and not pretend it"*.

Discussion The two keywords that summarize what developers—who represent the production unit—think of green software marketing, are integrity and transparency. The participants argue that selling green software should even be more attractive and could constitute a good marketing argument that differentiates the product from other providers, even if there is no direct pressure from the consumer to build more energy efficient software [21].

However, this marketing has to be very transparent towards both developers and end-users and avoid misleading communication and greenwashing.

> To answer RQ 1, we identified many hurdles that prevent a wider adoption of green software design, including the lack of knowledge, awareness, time, tools and communication.
>
> To answer RQ 2, we conclude that developers main requirements for tooling are the simplicity of interaction and integration with current procedures such as CI/CD, and global scores and KPIs to track the evolution of SEC.

3.4 Implications

We summarize the results of our study as sets of implications for developers, the company, tool creators, and researchers. We argue that these implications constitute a rich knowledge base that could guide understanding and promoting software energy consumption and green software design.

3.4.1 For Developers

Developers have been the data source of our study. Hence, we can retrieve a couple of implications for them:

- Given the observed level of awareness in our interviews, we suggest that developers should seek more information on environmental issues in general and the impact of the IT sector in particular. This would help the developers to grasp the importance of these issues and motivate them to work on them;

- Some developers seem to consider the changes implied by green software design as threats to their careers. We advocate instead that it could be an opportunity for professionals in the IT sector: skills in this emerging domain will be in short supply in the future while demands will probably increase substantially, especially with the growing concern about green technologies and energy consumption. Therefore, we encourage developers to invest time now in these key skills and argue that it will pay off quickly enough;

- Developers seem to be more receptive to messages coming from their peers. Therefore, developers with better knowledge about green software design should volunteer to help to inform and teach, both in their project team and to the whole IT community. Organizing presentations, submitting talks to developers conferences and frequently posting on the public and company social networks are effective ways of doing it;

3.4.2 For Decision Makers

We can extract many implications and responsibilities for decision-makers—a.k.a companies—from our results, including:

- The main role of the company is to maintain a large communication campaign about SEC that encompasses: *i)* ensuring a high level of transparency with the employees regarding the "green" vision and objectives, *ii)* running a long-term awareness program (with presentations for example), and *iii)* providing the developers with the necessary knowledge (through training programs, workshops, etc.);

- Developers described the identification of green objectives for development projects as one of the most efficient ways to motivate employees about green software design and clarify the company's position and engagements. These objectives would create additional motivation for developers, and would define entries that developers and product owners would discuss, validate and readjust, at every period;

- Developers also requested for necessary resources so they can achieve green software design objectives. The resources include the tools (and/or budget) that allow monitoring the software energy consumption and the necessary time to do it. Yet, there is already exploitable resources that the company could make use of, such as *i)* developers who already know green software design who could help in the communication/teaching process and *ii)* the ability of developers to adapt to different technologies, to attribute the human resource in the most convenient project;

- Marketing the green aspect of the software is also a major sector that the company should prioritize, as it would help to get clients' feedback regarding the market and the products. This would help to readjust the objectives and product specifications if necessary.

3.4.3 For Tool Makers

Our study provides numerous guides and specifications for tool creators:

- Developers ask for tools that can output a global score / KPI to summarize the energetic footprint of the source code. This score can then be used for communication with other stakeholders in the projects, to check that energy efficiency targets have been met, but also to easily follow the evolution of the energy consumption of software through commits, for example using graphical representations;
- At the same time, more advanced tools are also needed, that provide more detailed information to allow low-level diagnosis of the source code and identify the exact problems and solutions. This family of tools must however still be simple to use and provide graphical representations to simplify the interaction with the displayed information. Therefore, we argue that toolmakers must pay close attention to the usability of their tools, which is paramount to ensure that developers will be confident about using them;
- While there is at this time no clear consensus on the effectiveness of static code analysis for energy efficiency purposes, developers could be persuaded to use energy-focused linters, designed to flag bad practices and common "energy bugs" [115], as long as their benefits are demonstrated. To better convince developers of using such linters, toolmakers should concentrate on demonstrating the effectiveness of this approach, and integrate them in commonly used editors and IDE;
- The developers expect these tools to integrate seamlessly with available tool-chains, especially for *Continuous Integration* and *Continuous Delivery* (CI/CD), to automate the analysis and report on the energy footprint along with other metrics like code quality, performance, and tests reports.

3.4.4 For Researchers

Our study confirms previous work results—such as Pang et al. [114]—that highlighted the lack of knowledge and awareness among developers. Moreover, we present a new set of information that could be considered and extended in further research about green software design:

- The correlation between the technologies developers use and their level of awareness on green software design is an interesting research topic, like Manotas et al. [91] who stated that mobile developers are more concerned about software energy consumption problems. In our study, there seems to be no consensus among developers, even those coming from the same technical background. This implies there is no strong correlation between developers' awareness, and the technology they use. Some empirical studies with a larger population should be conducted to investigate the correlation between developers considerations and their background;

- We discuss a specific case study to understand the state of mind of a set of experienced developers in a large company of more than 100,000 employees in the telecommunication sector. However, it does not make our study automatically generalizable for other companies operating in a different sector, or companies of a different size, or even companies located in a different geographical area. Further empirical studies can be conducted on other samples, such as startups, collaborative projects, open-source projects, etc. across different regions and with numerous domains of activity. Quantitative studies are more adapted to conduct this kind of studies with much bigger samples to track the different axes that may change developers' opinions and constraints regarding green software design;

- Our study highlights the need for KPIs/scores about software energy consumption. Hence, considerable research work is still needed to build the theoretical knowledge on the right set of KPIs and visualization formats that tool creators could implement, and that would speak better to both developers and decision-makers;

- We noticed during our study that a couple of trade-offs should be considered along with green software design. Not all developers fully distinguish software energy consumption from its performance, thus multiple works that could help developers to make choices and to deal with the slight difference when it occurs can still be done. Moreover, the participants foresee a trade-off between the development time and using less consuming programming languages which should be proven or denied in further research.

3.5 Threats To Validity

A couple of issues may affect the validity and the generalizability of our work. First, our population might not be representative of all kinds of populations for several reasons. Starting with the sample size, which offered a certain level of data saturation [135, 145], but is still not quite large for better generalizability to other populations. Conducting, transcribing, and analyzing the interviews are tedious manual tasks that are very time and energy-consuming. Hence, considering a much bigger sample size would have massively increased the workload for a qualitative study and would have implied considering diverse company types, regions, participants' profiles, etc. This is in contrast to our study's purpose to deliver a concrete case study to understand and promote green software design within a company. Moreover, considering a population of only experts and experienced developers that are likely more interested in the topic than average is not the best representation of all scenarios, as junior developers could have caused a slower data saturation, for example. The purpose behind selecting experienced and expert developers with a certain level of awareness is however to conduct a study that delivers insightful and high-quality findings and implications. This would highlight the most relevant issues and build a decent strategy to promote green software design within the company, even if it does not make it trivially generalizable. Another possible threat is the specificity of the region. All of our participants are from the same country and the same company. While this is useful to build an understanding of developers' opinions within the same company, our study may have missed other opinions from other countries/regions.

Other issues may be related to the qualitative study process such as the validity of our participants' answers. We tried to avoid *"yes or no"* questions and we encouraged our participants to argue on their opinions so we can have more details and alleviate any misleading expressions or misunderstanding/misinterpretation from us. Unfortunately, most of our participants were located on remote sites. Hence, we conducted those interviews through calls, which does not allow capturing as many details as live interviews, such as the interviewee's behavior and gestures. Moreover, our analysis and interpretation can be a threat to the validity of our findings. To alleviate this issue, we were two persons to code and analyze the data separately so we can offer more credible results.

3.6 Summary

Our study of the state of the art revealed the lack of a full qualitative study that conducts an in-depth analysis of practitioners' requirements and comprehensions. Most of the studies we encountered focused on presenting a survey of what developers know about green software design. While this is very important, it does not deliver a better understanding of what are the developer's requirements and how to get his/her attention to the problem.

This chapter rather conducts an in-depth qualitative study about software energy consumption considerations among a population of experienced and expert developers in a large company. This constitutes a solid case study that exposes key implications to be considered within the company, but also preliminary findings that could be checked across other companies' profiles. Our study investigates 3 major questions: 1) What do our participants know about software energy consumption? 2) what are the main hurdles that prevent green software design considerations? including the main specifications for tooling (or tool set) that matches developers' expectations and experience with other software metrics, such as CPU consumption or execution time? 3) What are the most efficient ways to reach the developers and the deciders, and keep them aware of the importance of energy considerations in software development and their role to promote it? The answer to the first question confirms the results of the many papers[114, 119] regarding the moderate awareness and lack of knowledge of developers on green software design. This also means that in at least 6 years, the situation has not really evolved and developers are still struggling to handle software energy consumption issues. Moreover, our participants reported on a very mitigated consideration of static code analysis as it does not consider the execution environment, in contrast to the results of Manotas et al. [91].

The novelty exposed in this chapter is mainly related to the second and third questions, in which we seek to understand developers' requirements to better digest green software design and include it in their development routines and considerations. We highlight many points that should be enhanced to achieve a certain level of maturity of SEC considerations, such as: 1) setting individual and global objectives about green software design, 2) encouraging presentations and training to raise awareness and remedy to lack of knowledge, 3) including SEC in projects planning with dedicated time, tools and budget, and 4) developing and using dedicated tools, which must facilitate the integration of SEC considerations in the developers' daily activity and whose specifications can be drafted from this work.

Chapter 4

Controlling the Measurement of Energy Consumption

One of the most discussed issues during the interviews of the qualitative study is the lack of tools, especially tools and ways that allow to accurately measure the software energy consumption and ensure the reproducibility of experiments/measures. Indeed, measuring the energy consumption of a software system remains a tedious task for practitioners. In particular, the energy measurement process may be subject to a lot of variations that hinder the relevance of potential comparisons. While the state of the art mostly acknowledges the impact of hardware factors (chip printing process, CPU temperature, etc.), this chapter investigates the impact of controllable factors on these variations. More specifically, we conduct an empirical study of multiple controllable parameters that one can easily tune to tame the energy consumption variations when benchmarking software systems.

The main factors we studied encompass: experimental protocol, CPU features (C-states, Turbo Boost, core pinning) and generations, as well as the operating system. Our experiments showed that, for some workloads, it is possible to tighten the energy variation by up to 30×. Finally, we summarize our results as guidelines to tame energy consumption variations and conduct accurate energy consumption measurements.

The remainder of this chapter is organized as follows. Section 4.1 gives an overview on the chapter Section 4.2 formalizes our research questions. Section 4.3 reports on the experimental setup (hardware, benchmarks, tools and methodology) we used in this work. Section 4.4 analyzes the causes of the variations we observed during experiments. Finally, we discuss the results of the different experimented factors, and their impact on the energy variation in Section 4.5.

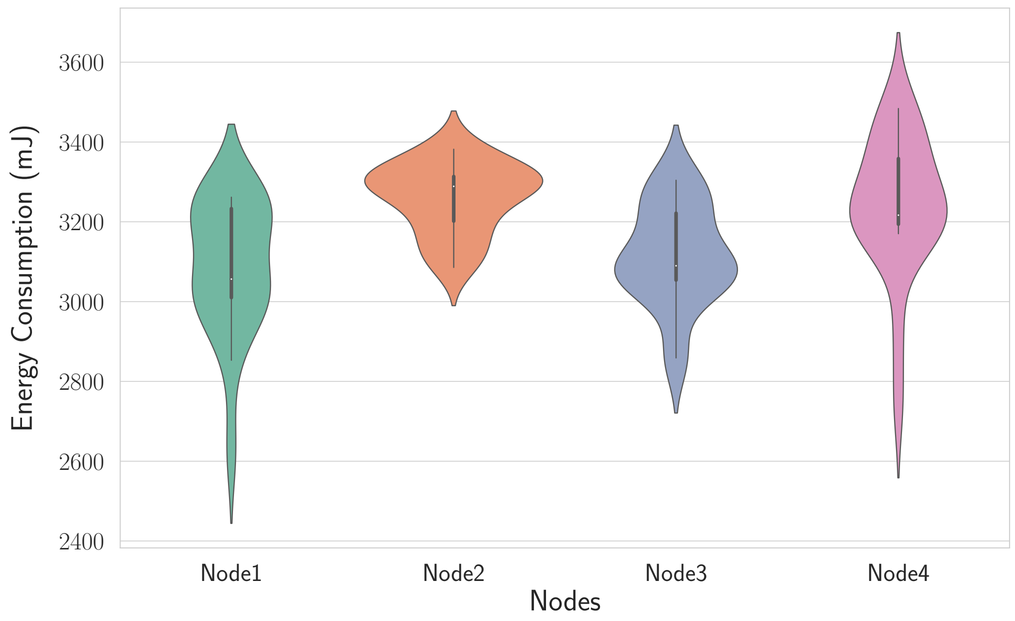


Figure 4.1: CPU energy variation for the benchmark CG

4.1 Overview

To conduct robust evaluations, practitioners often try to ensure reproducible environmental conditions in order to properly benchmark their software systems. In this area, reproducibility might be achieved by ensuring the same execution settings of physical nodes, virtual machines, clusters or cloud environments. Recently, the research community has been investigating typical "crimes" in systems benchmarking and established guidelines for conducting robust and reproducible evaluations [142].

In theory, using identical CPU, same memory configuration, similar storage and networking capabilities, should favour reproducible experiments. However, when it comes to measuring the energy consumption of a system, applying acknowledged guidelines and carefully repeating the same benchmark can nonetheless lead to different energy footprints not only on homogeneous nodes, but even within a single node. This difference—also called *energy variation* (EV)—has become a serious threat to the accuracy of experimental evaluations.

Figure 4.1 illustrates this variation problem as a violin plot of 20 executions of the benchmark *Conjugate Gradient* (CG) taken from the *NAS Parallel Benchmarks* (NBP) suite [14], on 4 nodes of an homogeneous cluster (the cluster Dahu described in Table 4.1) at 50 % workload. One can observe a large variation of the energy consumption, not only among homogeneous nodes, but also at the scale of a single node, reaching up to 25 % in this example.

Most of the state of the art has been investigating this power consumption issue from a hardware perspective [24, 141] and reported that the causes of such energy variations are CMOS manufacturing process of transistors in a chip, differences in node assembly and data center hot spots. Additionally, Heinrich et al. [53] described it as a combination of parameters, mentioning a list of candidate factors, such as the thermal effect or the CPU frequency, but did not to deliver a deeper analysis of these factors. Unfortunately, not all hardware factors can be tuned to tame the energy variation that can be observed in experiments. For example, managing the CPU temperature or a server position in a cluster, are not actions that one can easily do, especially on the modern data centers and cloud platforms. Therefore, the goal of this chapter is to investigate the spectrum of factors that can cause or increase the variability of energy consumption in experiments and systems benchmarking, and to propose effective guidelines to control such factors in order to mitigate this variability. While this chapter does not question the benefits of established CPU features, like C-states or Turbo Boost, it delivers a deeper analysis of the effects they may introduce on a wide set of experiments and nodes. By quantifying potential energy variations induced by controllable factors, we intend to identify the proper configurations that minimize energy variations, depending on the workload characteristics. These guidelines aim at supporting practitioners in conducting more accurate experiments and reporting reproducible energy consumption.The key contributions of this chapter can therefore be summarized as:

1. Providing a better understanding of the energy variation, by using different generations of CPU deployed in 4 clusters with more than 100 physical nodes, and by considering existing systems benchmarks with diverse workloads;
2. Identifying controllable factors that contribute to the variation in CPU energy consumption, comparing them against the state of the art, and completing them with other uncovered assumptions;
3. Reporting on some guidelines on how to conduct reproducible experiments with less energy variations;
4. Discussing the differences between inter-nodes and intra-nodes energy variations.

4.2 Research Questions

Part of the energy consumption variation is due to chip manufacturing differences or some of the previously discussed enforced factors, such as the thermal effect or the servers placement. Those parameters are often tricky to manage, as we cannot have a perfect chips manufacturing process, or assume that two identical processors have the same thermal behavior. We will therefore focus on providing the practitioners with an empirical study of some controllable parameters that can be tuned to conduct experimental evaluations of the energy consumption

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

Table 4.1: Description of clusters included in the study

with less variation. Especially if the practitioners do not have physical access to the data center or BIOS configuration, which is the case on most of the modern cloud platforms and data centers. These parameters span the choice of the benchmarking protocol, processor frequencies management, operating system tuning or some other parameters. Heinrich et al. [53] mentioned some of these potential parameters.

In this chapter, we will therefore investigate the following controllable factors, which we formalize as 4 research questions:

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

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

RQ 3: Does the choice of the processor matter to mitigate the energy variation? and finally

RQ 4: What is the impact of the operating system on the energy variation?

4.3 Experimental Setup

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

4.3.1 Hardware Platform

We considered 4 distinct clusters of variable sizes and different generations of CPU, as summarized in Table 4.1. In particular, we used the Grid5000 (G5K) platform in our experiments [16, 94]. G5K is a bare metal cloud platform that can be used to provision clusters of identical nodes. In our study, we mainly used the cluster Dahu located in Grenoble to run most of our tests, as it has one of the newest Xeon CPUs. We also used the clusters Chetemi, Ecotype and Paranoia in some of our experiments. Table 4.1 describes the configurations of the clusters we considered.

As most of the nodes are equipped with two sockets (physical processors), we use the acronym CPU or socket to designate one of the two sockets and PU for the operating system *processing unit*. The number of PU often doubles the number of available cores because of the hyper-threading support, as the OS considers 2 hyper-threads sharing the same core as 2 different PU. Figure 4.2 illustrates a detailed topology of a node belonging to the cluster Dahu.

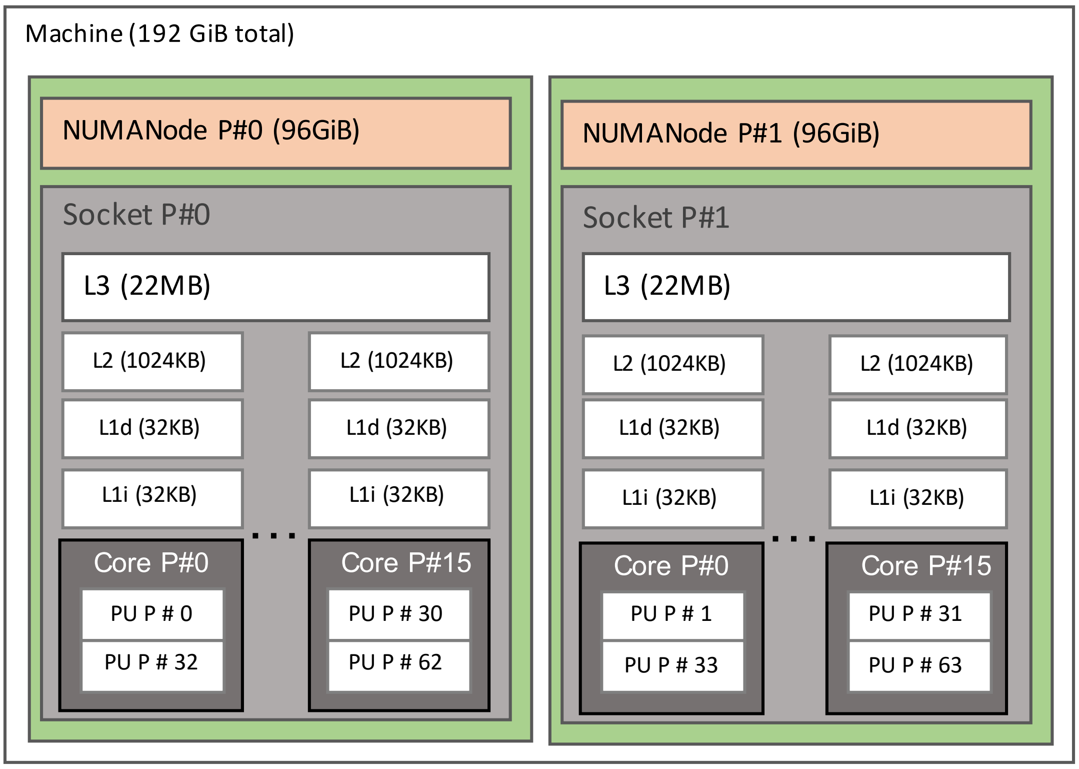


Figure 4.2: Topology of the nodes of the cluster Dahu

4.3.2 Systems Benchmarks

Our first criterion to choose the systems benchmarks was the scalability, as we need to run tests at different workloads by choosing the right number of used PU. The other criteria are the documentation, the accuracy and the references to the benchmark. *NAS Parallel Benchmark* (NPB v3.3.1) [14] is one of the most used suite of benchmarks in the HPC literature and it fulfills our benchmarking requirements. We mainly used the pseudo application *Lower-Upper symmetric Gauss-Seidel* (LU), the *Conjugate Gradient* (CG) and *Embarrassingly Parallel* (EP) computation-intensive benchmarks in our experiments, with the C data class. These are the main benchmarks used in many similar works [53]. Nonetheless, in order to validate our results on a wider set of benchmarks and applications, we also used `Stress-ng v0.10.0`,[1] `pbzip2 v1.1.9`,[2] `linpack`[3] and `sha256 v8.26`[4] as representative systems benchmarks to conduct our experiments with a broad diversity of workloads.

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

4.3.3 Measurement Tools and Methodology

To study the energy consumption of nodes, we considered Intel RAPL [37, 67], which is one of the most accurate tools to report the CPU/DRAM global energy consumption. We also used POWERAPI [32], which is a power monitoring toolkit that builds a model over RAPL to compute the energy consumption at process-level when we needed to isolate energy consumption of a single process. Our clusters are provisioned with a minimal version of Debian 9 (4.9.0 kernel version) where we installed Docker (version 18.09.5), which will be used to run the RAPL sensor and the benchmark itself. The energy sensor collects RAPL reports and stores them in a remote MONGODB instance, allowing us to perform *post-mortem* analysis in a dedicated environment. Using Docker makes the deployment process easier on the one hand, and provides us with a built-in control group encapsulation of the conducted tests on the other hand. This allows POWERAPI to measure all the running containers, even the RAPL sensor consumption, as it is isolated in a container. One potential threat covers the impact of Docker on the energy variation. We therefore conducted a preliminary experiment by running the same benchmarks LU, CG and EP in a Docker container and a flat binary format on 3 nodes of the cluster Dahu to assess if Docker induces an additional variation. Figure 4.3 reports that this is not the case, as the energy consumption variation does not get noticeably affected by Docker while running a same compiled version of the benchmarks at 5 %, 50 % and 100 % workloads. In fact, while Docker increases the energy consumption due to the extra layer it implements [39], it does not noticeably affect the energy variation. The STD is even slightly smaller ($STD_{Docker} = 192mJ, STD_{Binary} = 207mJ$).

Every experiment is conducted on 100 iterations, on multiple nodes and using the 3 NPB benchmarks we mentioned, with a warmup phase of 10 iterations for each experiment. In most cases, we were seeking to evaluate the STD, which is the most representative factor of the energy variation. We tried to be very careful, while running our experiments, not to fall in the most common benchmarking "crimes" [142]. As we study the STD difference of measurements we observed from empirical experiments, we use the bootstrap method [40] to randomly build multiple subsets of data from the original dataset, and we draw the STD density of those sets, as illustrated in Figure 4.3.

We mainly consider 3 different workloads in our experiments: single process, 50 %, and 100 %, to cover the low, medium and high CPU usage when analyzing the studied parameters effect, respectively. These workloads reflect the ratio of used PU count to the total available PU.

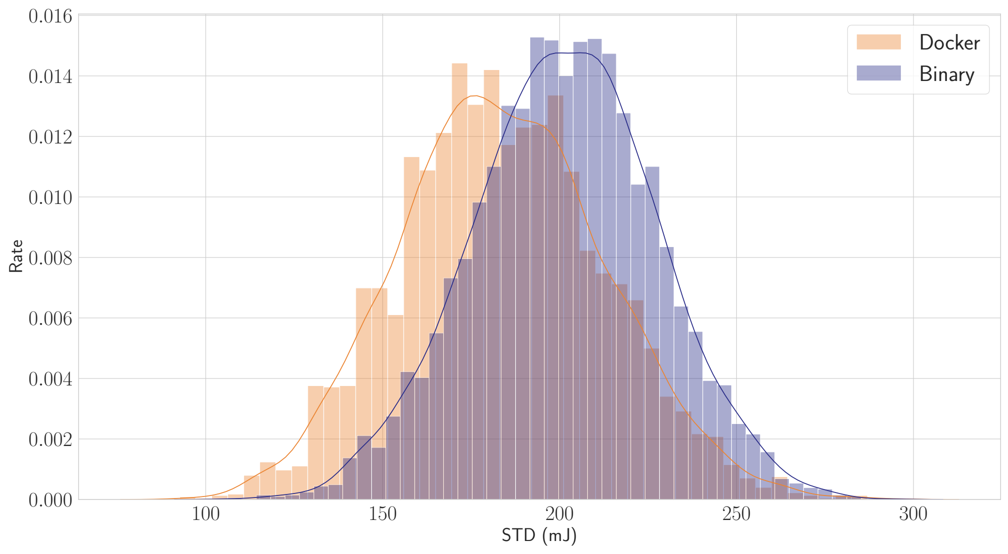


Figure 4.3: Comparing the variation of binary and Docker versions of aggregated LU, CG and EP benchmarks

4.4 Energy Variation Analysis

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

4.4.1 RQ 1: Benchmarking Protocol

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

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

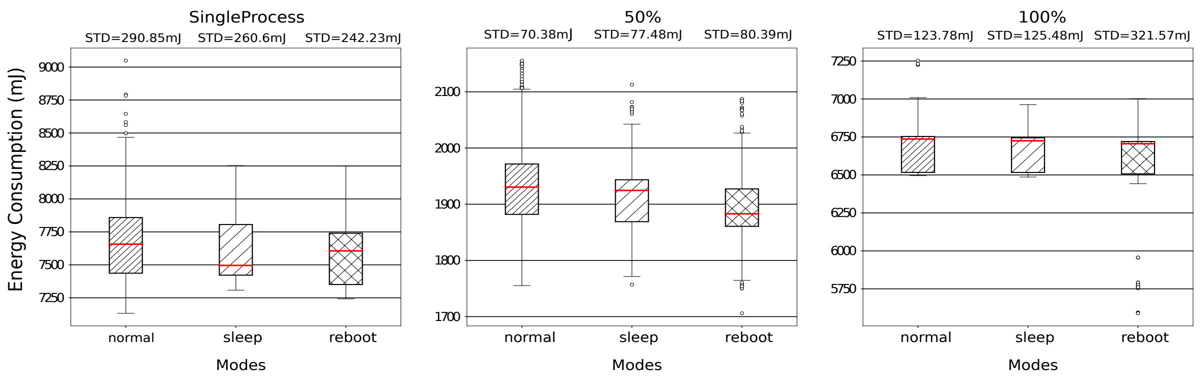


Figure 4.4: Energy variation with the normal, sleep and reboot modes

could have an impact on the energy variation. The reboot mode, on the other hand, is the most straightforward way to reset the machine state after every iteration. It could also be beneficial to reset the CPU frequency and temperature, the stored data, the cache or the CPU registries. However, the reboot task takes a considerable amount of time, so rebooting the node after every single operation is not the fastest nor the most eco-friendly solution, but it deserves to be checked to investigate if it effectively enhances the overall energy variation or not.

Figure 4.4 reports on 300 aggregated executions of the benchmarks LU, CG and EP, on 4 machines of the cluster Dahu (cf. Table 4.1) for different workloads. We note that the results have been executed with different datasets sizes (B, C and D for single process, 50 % and 100 % respectively) to remedy to the brief execution times at high workloads for small datasets. This justifies the scale differences of reported energy consumptions between the 3 modes in Figure 4.4. As one can observe, picking one of these strategies does not have a strong impact on the energy variation for most workloads. In fact, all the strategies seem to exhibit the same variation with all the workloads we considered—*i.e.*, the STD is tightly close between the three modes. The only exception is the reboot mode at 100 % load, where the STD is 150 % times worse, due to an important amount of outliers. This goes against our expectation, even when setting a warm-up time after reboot to stabilize the OS.

In Figure 4.5, we study the standard deviation of the three modes by constituting 5,000 random 30-iterations sets from the previous executions set and we compute the STD in each case, considering mainly the 100 % workload as the STD was 150 % higher for the reboot mode with that load. We can observe that the considerable amount of outliers in the reboot mode is not negligible, as the STD density is clearly higher than the two other modes. This makes the reboot mode less appropriate for the energy variation at high workloads.

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

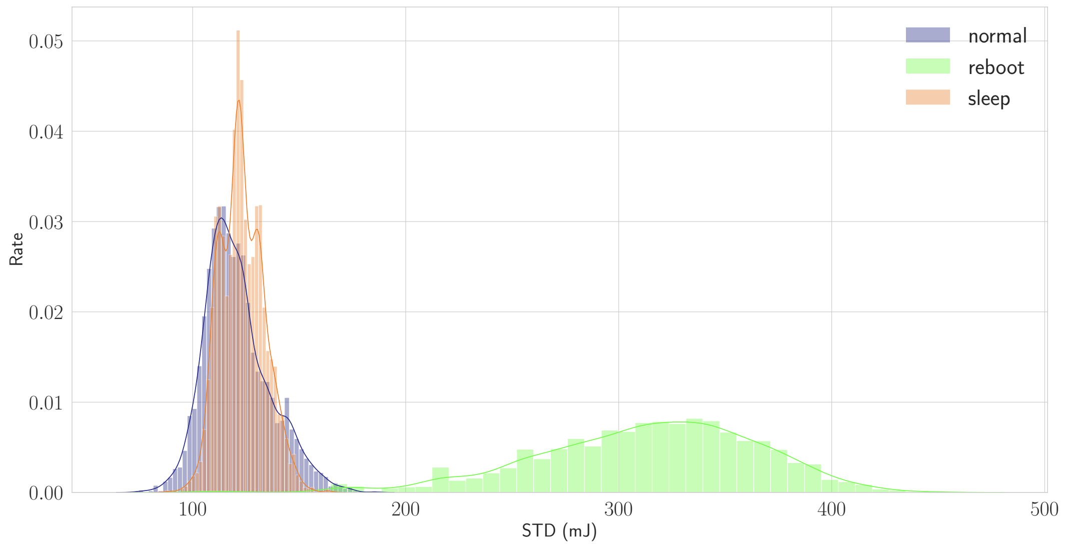


Figure 4.5: STD analysis of the normal, sleep and reboot modes

4.4.2 RQ 2: Processor Features

The C-states provide the ability to switch the CPU between more or less consuming states upon activities. Turning the C-states on or off has been subject of many discussions [144] because of its dynamic frequency mechanism, but, to the best of our knowledge, there have been no fully conducted C-states behavior analysis on CPU energy variation.

We intend to investigate how much the energy consumption varies when disabling the C-states (thus, keeping the CPU in the C0 state) and at which workload. Figure 4.6 depicts the results of the experiments we executed on three nodes of the cluster Dahu. On each node, we ran the same set of benchmarks with two modes: C-states on, which is the default mode, and C-states off. Each iteration includes 100 executions of the same benchmark at a given workload, with three workload levels. We note that our results have been confirmed with the benchmarks LU, CG and EP.

We can clearly see the effect that has the C-states off mode when running a single-process application/benchmark. The energy consumption varies 5 times less than the default mode. In this case, only one CPU core is used among 2×16 physical cores. The other cores are switched to a low consumption state when C-states are on, the switching operation causes an important energy consumption difference between the cores, and could be affected by other activities, such as the kernel activity, causing a notable energy consumption variation. On the other hand, switching off the C-states would keep all the cores—even the unused ones—at a high frequency usage. This highly reduces the variation, but causes up to 50 % of extra energy consumption in this test ($Mean_{C-states-off} = 11{,}665 mJ, Mean_{C-states-on} = 7{,}641 mJ$).

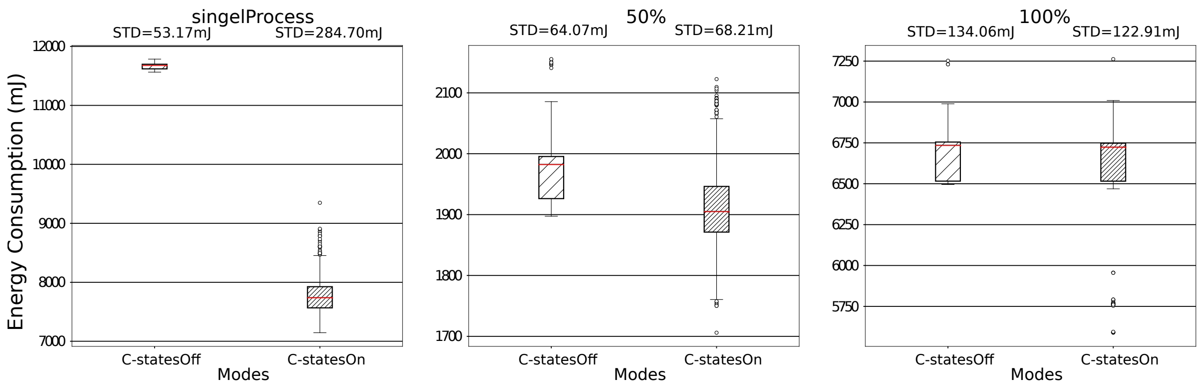


Figure 4.6: Energy variation when disabling the C-states

At a 100 % workload, disabling the C-states seems to have no effect on the total energy consumption nor its variation. In fact, all the cores are used at 100 % and the C-states module would have no effect, as the cores are not idle. The same reason would apply for the 50 % load, as the hyper-threading is active on all cores, thus causing the usage of most of them. This leads to mainly two questions: Can a process pinning method reduce/increase the energy variation? And, how does the energy consumption variation evolve at different PU usage levels?

Cores Pinning

To answer the first question, we repeated the previous test at 50 % workload. In this experiment, we considered three cores usage strategies, the first one (S1) would pin the processes on all the PU of one of the two sockets (including hyper-threads), so it will be used at 100 %, and leave the other CPU idle. The second strategy (S2) splits the workload on the two sockets so each CPU will handle 50 % of the load. In this strategy, we only use the core PU and not the hyper-threads PU, so every process would not share its core usage (all the cores are being used). The third strategy (S3) consists also on splitting the workload between the two sockets, but considering the usage of the hyper-threads on each core—*i.e.*, half of the cores are being used over the two CPU. Figure 4.7 reports on the energy consumption of the three strategies when running the benchmark `CG` on the cluster `Dahu`. We can notice the big difference between these three execution modes that we obtained only by changing the PU pinning method (that we acknowledged with more than 100 additional runs over more than 30 machines and with the benchmarks `LU` and `EP`). For example, S2 is the least power consuming strategy. We argue that the reason is related to the isolation of every process on a single physical core, reducing the context switch operations. In the first and third strategy, 32 processes are being scheduled on 16 physical cores using the hyper-threads PU, which will introduce more context switching, and thus more energy consumption.

We note that even if the first and third strategies are very similar (both use hyper-threads, but only on one CPU for the first and on two CPU for the third), the gap between them is

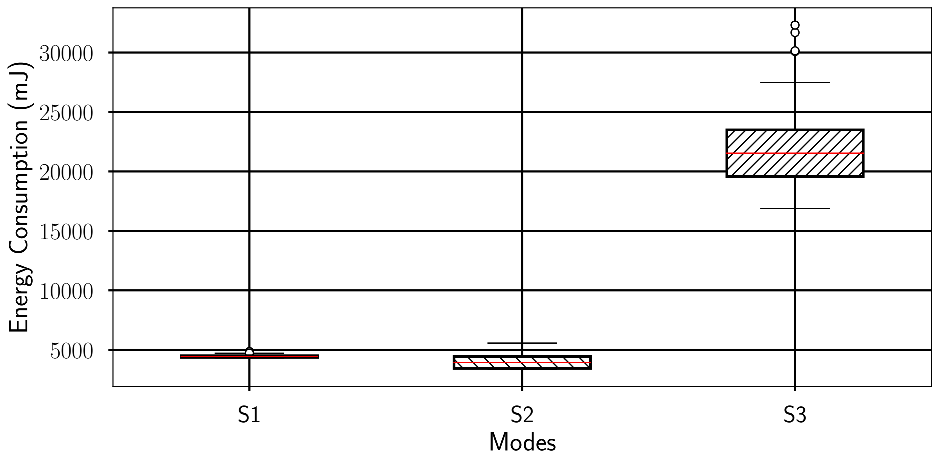


Figure 4.7: Energy variation considering the three cores pinning strategies at 50 % workload

considerable variation-wise, as the variation is 30 times lower in the first strategy ($STD_{S1} = 116mJ$,$STD_{S3} = 3{,}452mJ$). This shows that the usage of the hyper-threads technology is not the main reason behind the variation, the first strategy has even less variation than the second one and still uses the hyper-threading.

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

Our hypothesis regarding the worst results that we observed when using the third strategy is the recurrent context switching, added to the OS scheduling that could reschedule processes from a socket to another, which invalids the cache usage as a process can not take profit of the socket local L3 cache when it moves from a CPU to another (cf. Figure 4.2).

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

Table 4.2 reports on the average results for the STD comparison on four other nodes of the cluster Dahu at 50 %, with the benchmarks LU, CG and EP. In fact, the CPU usage strategy S1 is by far the experimentation mode that gave the least variation. The STD is almost 5 times better

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

Table 4.2: STD (mJ) comparison for 3 pinning strategies

than the strategy S2, but is up to 10 % more energy consuming ($Mean_{S1} = 4469mJ$, $Mean_{S2} = 4016mJ$). On the other hand, the strategy S3 is the worst, where the energy consumption can be up to 5 times higher than the strategy S2 ($Mean_{S2} = 4016mJ$, $Mean_{S3} = 21645mJ$) and the variation is much worst (30 times compared to the first strategy). These results allow us to have a better understanding of the different processes-to-PU pinning strategies, where isolating the workload on a single CPU is the best strategy. Using the hyper-threads PU on multiple sockets seems to be a bad recommendation, while keeping the hyper-threading enabled on the machine is not problematic, as long as the processes are correctly pinned on the PU. Our experiments show that running one hyper-thread per core is not always the best to do, at the opposite of the claims of Marathe et al. [93].

Processes Threshold

To answer the second question regarding the evolution of the energy variation at different levels of CPU usage, we varied the used PU's count to track the EV evolution. Figure 4.8 compares the aggregated energy variation when the C-states are on and off using 2, 4 and 8 processes for the benchmarks LU, CG and EP. This figure confirms that disabling the CPU C-states does not decrease the variation for all the workloads.When running only 2 processes, turning off the C-states reduces the STD up to 6 times, but consumes 20 % more energy ($Mean_{C-states-on} = 10{,}334mJ$, $Mean_{C-states-off} = 12{,}594mJ$). This variation is 4 times lower when running 4 processes and almost equal to the C-states on mode when running 8 processes. In fact, running more processes implies using more CPU cores, which reduces the idle cores count, so the cores will more likely stay at a higher consumption state even if the C-states mechanism is on.

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

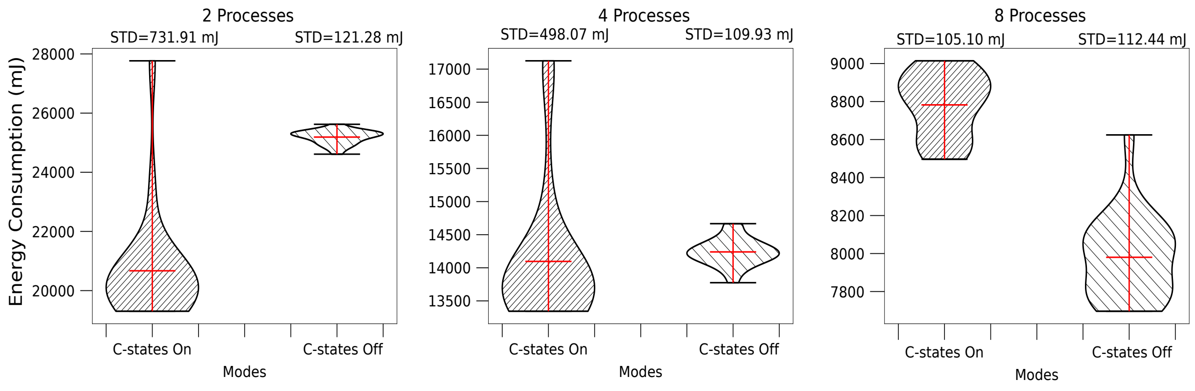


Figure 4.8: C-states effect on the energy variation, regarding the application processes count

Turbo Boost

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

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

The experiment we conducted showed that disabling the Turbo Boost does not exhibit a trivial positive or negative effect on the energy variation. Table 4.3 compares the STD when enabling/disabling the Turbo Boost, where the columns are a combination of workload and benchmark. In fact, we only got some minor measurements differences when switching on and off the Turbo Boost, and were in favor or against the usage of the Turbo Boost while repeating tests, considering multiple nodes and benchmarks. This behavior is mainly related to the *thermal design power* (TDP), especially at high workloads. When a CPU is used at its maximum capacity, the cores would be heating up very fast and would hit the maximum TDP limit. In this case, the Turbo Boost cannot offer more power to the CPU because of the CPU thermal restrictions. At lower workloads, the tests we conducted showed that the Turbo Boost impact on energy variation is not trivial or predictable. We cannot affirm that the Turbo Boost does not have an impact on all the CPU, as we only tested on two recent Xeon CPU (clusters Chetemi

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

Table 4.3: STD (mJ) comparison when enabling/disabling the Turbo Boost

Cluster	Dahu	Chetemi	Ecotype	Paranoia
Arch	Skylake	Broadwell	Broadwell	Ivy Bridge
Freq	2.1 GHz	2.2 GHz	1.8 GHz	2.2 GHz
TDP	125 W	85 W	55 W	95 W
5%	364	210	**75**	**76**
50%	98	86	**49**	244
100%	119	116	**106**	240

Table 4.4: STD (mJ) comparison of experiments from 4 clusters

and Dahu). We confirmed our experiments on these machines 100 times at 5 %, 25 %, 50 % and 100 % workloads.

We conclude that CPU features **highly impact** the energy variation as an answer for RQ 2. Especially by disabling the C-states for low workloads and correctly pining the processes on the available PU.

4.4.3 RQ 3: Processor Generation

Intel microprocessors have noticeably evolved during these last 20 years. Most of the new CPU come with new enhancements to the chip density, the maximum Frequency or some optimization features like the C-states or the Turbo Boost. This active evolution caused different generations of CPU to handle a task differently. The aim of this experiment is not to justify the evolution of the variation across CPU versions/generations, but to observe if the user can choose the best node to execute his/her experiments. Previous papers have discussed the evolution of the energy consumption variation across CPU generations and concluded that the variation is getting higher with the latest CPU generations [93, 149], which makes measurements stability even worse. In this experiment, we therefore compare four different generations of CPU with the aim to evaluate the energy variation for each CPU and its correlation with the generation. Table 4.4 indicates the characteristics of each of the tested CPU.

Table 4.4 also shows the average energy variation of the different generations of nodes for the benchmarks LU, CG and EP. The results attest that the latest versions of CPU do not necessarily cause more variation. In the experiments we ran, the nodes from the cluster Paranoia tend to cause more variation at high workloads, even if they are from the latest generation,

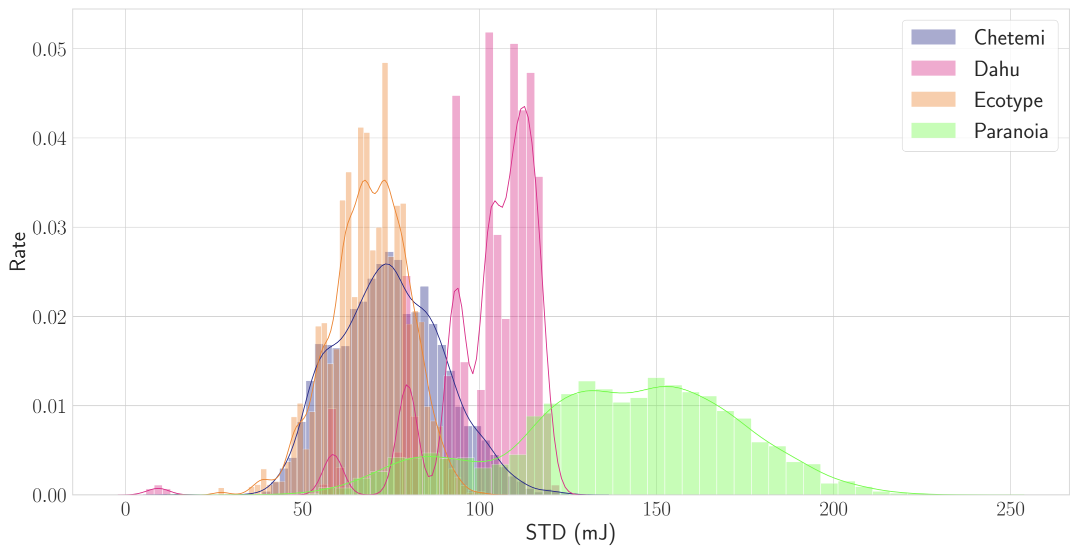


Figure 4.9: Energy consumption STD density of the 4 clusters

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

Figure 4.9 is an illustration of the aggregated STD density of more than 5,000-random values sets taken from all the conducted experiments. This shows that the cluster Paranoia reports the worst variation in most cases, and that Ecotype is the best cluster to consider to get the least variations, as it has a higher density for small variation values.

We conclude on **affirming RQ 3**, as selecting the right CPU can help to get less variations.

4.4.4 RQ 4: Operating System

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

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

Inspecting the OS idle energy variation is not sufficient to relate the energy variation to the active job. In fact, the OS can behave differently regarding the resource usage when running a task. To evaluate the OS and the job energy consumption separately, we used the POWERAPI toolkit. This fine-grained power meter allows the distribution of the RAPL global energy across all the Cgroups of the OS using a power model. Thus, it is possible to isolate the job energy consumption instead of the global energy consumption delivered by RAPL. To do so, we ran tests with a single process workload on the cluster Dahu, and used the POWERAPI toolkit to measure the energy consumption. Then, we compared the job energy consumption to the global RAPL data. We calculated the Pearson correlation [2] of the energy consumption and variation between global RAPL and POWERAPI, as illustrated in Figure 4.10. The job energy consumption and variation are strongly correlated with the global energy consumption and variation with the coefficients 93.6 % and 85.3 %, respectively. However, this does not completely exclude the OS activity, especially if the jobs have tight interaction with the OS through the signals and system calls. This brings a new question on whether applying extra-tuning on a minimal OS would reduce the variation? As well as what is the effect of the Meltdown security patch—that is known to be causing some performance degradation [70, 82]—on the energy variation?

OS Tuning

An OS is a pack of running processes and services that might or not be required for its execution. In fact, even using a minimal version of a Debian Linux, we could list many OS running services and processes that could be disabled/stopped without impacting the test execution. This extra-tuning may not be the same depending on the nature of the test or the OS. Thus, we conducted a test with a deeply-tuned OS version. We disabled all the services/processes

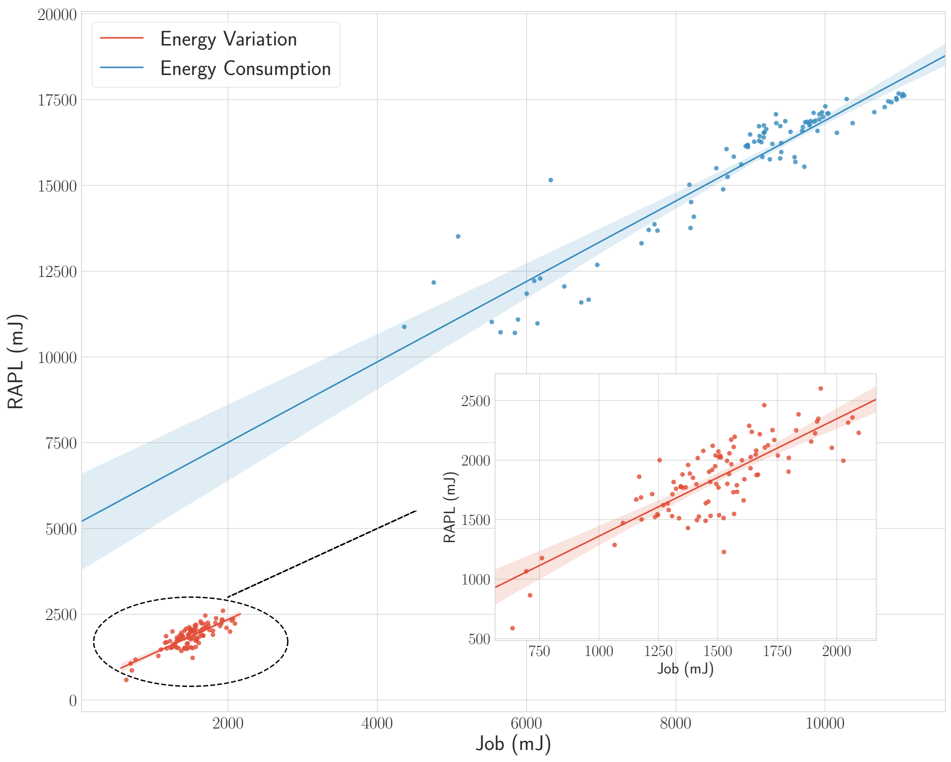


Figure 4.10: The correlation between the RAPL and the job consumption and variation

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

Table 4.5: STD (mJ) comparison before/after tuning the OS

that are not essential to the OS/test running, including the OS networking interfaces and logging modules, and we only kept the strict minimum required to the experiment's execution. Table 4.5 reports on the results for running single process measurements with the benchmarks CG, LU and EP, on three servers of the cluster Dahu, before and after tuning the OS. Every cell contains the *STD* value before the tuning plus/minus a ratio of the energy variation after the tuning. We notice that the energy variation varies less than 10 % after the extra-tuning. We also notice that this variation is not stable from a node to another. Moreover, 10 % of variation is not a representative difference, due to many factors that can affect it such as the CPU temperature or the measurement errors.

Speculative Executions

Meltdown and Spectre are two of the most famous hardware vulnerabilities discovered in 2018, and exploiting them allows a malicious process to access others processes data that is supposed to be private [70, 82]. They both exploit the speculative execution technique where a process anticipates some upcoming tasks, which are not guaranteed to be executed, when extra resources are available, and revert those changes if not. Some OS-level patches had been applied to prevent/reduce the criticality of these vulnerabilities. On the Linux kernel, the patch has been automatically applied since the version 4.14.12. It mitigates the risk by isolating the kernel and the user space and preventing the mapping of most of the kernel memory in the user space. Simakov et al. have studied [132] the impact of patching the OS on the performance. The results showed that the overall performance decrease is around 2–3 % for most of the benchmarks and real-world applications, only some specific functions exhibited a high performance decrease. In our study, we are interested in the applied patch's impact on the energy variation, as the performance decrease could induce an energy consumption increase. Thus, we ran the same benchmarks LU, CG and EP on the cluster Dahu with different workloads, using the same OS, with and without the security patch. Table 4.6 reports on the STD values before disabling the security patch. A minus means that the energy varies less without the patch being applied, while a plus means that it varies more. These results help us to conclude that the security patch's effect on the energy variation is not substantial and can be absorbed through the error margin for the tested benchmarks. In fact, the best case to consider is the benchmark LU where the energy variation is less than 10 % when we disable the security patch, but this difference is still moderate and not stable from a node to another due to other factor's impact such as CPU temperature. The little performance difference [70, 82] may only be responsible for a small variation, which will be absorbed through the measurement tools and external noise error margin in most cases.

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

Table 4.6: STD (mJ) comparison with/without the security patch

To answer RQ 4, we conclude that the OS **should not be the main focus** of the energy variation taming efforts.

4.5 Experimental Guidelines

To summarize our experiments, we provide some experimental guidelines in Table 4.7, based on the multiple experiments and analysis we did. These guidelines constitute a set of minimal requirements or best practices, depending on the workload and the criticality of the energy measurement precision. It therefore intends to help practitioners in taming the energy variation on the selected CPU, and conduct experiments with the least variations.

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

Table 4.7: Experimental Guidelines for Energy Variations

Table 4.7 gives a proper understanding of known factors, like the C-states and its variation reduction at low workloads. However, it also lists some new factors that we identified along the analysis we conducted in Section 4.4, such as the results related to the OS or the reboot mode. Some of the guidelines are more useful/efficient for specific workloads, as shown in our experiments. Thus, qualifying the workload before conducting the experiments can help in choosing the proper guidelines to apply. Other studied factors have not been mentioned in the guidelines, like the Turbo Boost or the Speculative execution, due to the small effect that has been observed in our study.

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

Figure 4.11 and 4.12 highlight the improvement brought by the adoption of our guidelines. They demonstrate the intra-node STD reduction at low and medium workloads for all the benchmarks used at different levels. Concretely, for low workloads, the energy variation is 2–6 times lower after applying the optimization guidelines for the benchmarks LU and EP, as well as LINPACK, while it is 1.2–1.8 times better for `Sha256`. For this workload, the overall energy consumption after optimization can be up to 80 % higher due to disabling the C-states that keeps all the unused cores at a high power consumption state ($Mean_{LU-normal-Dahu2} = 11{,}500mJ$, $Mean_{LU-optimized-Dahu2} = 20{,}508mJ$). For medium workloads, the STD, and thus variation, is up to 100 % better for the benchmark CG, 20–150 % better for the `pbzip2` application and up to 100% for STRESS-NG. We note that the optimized version consumes even less energy thanks to an appropriate core pinning method.

Figures 4.11 and 4.12 also highlight that applying the guidelines does not reduce the inter-nodes variation. This variation can be up to 30 % in modern CPU [149]. However, taming the intra-node variation is a good strategy to identify more relevant mediums and medians, and then perform accurate comparisons between nodes variation. Even though using the same node is always better, to avoid the extra inter-nodes variation and thus improve the stability of measurements.

4.6 Threats to Validity

A number of issues affect the validity of our work. For most of our experiments, we used the Intel RAPL tool, which has evolved along Intel CPU generations to be known as one of the most accurate tools for modern CPU, but still adds an important overhead if we adopt a sampling at high frequency. The other fine-grained tool we used for measurements is POWERAPI. It allows one to measure the energy consumption at the granularity of a process or a Cgroup by dividing the RAPL global energy over the running processes using a power model. The usage of POWERAPI adds an error margin because of the power model built over RAPL. Moreover, it does not offer a way to separate the static and dynamic power consumption. The RAPL tool mainly measures the CPU and DRAM energy consumption. However, even running CPU/RAM intensive benchmarks would keep a degree of uncertainty concerning the hard disk and networking energy consumption. In addition, the operating system adds a layer of confusion and uncertainty.

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

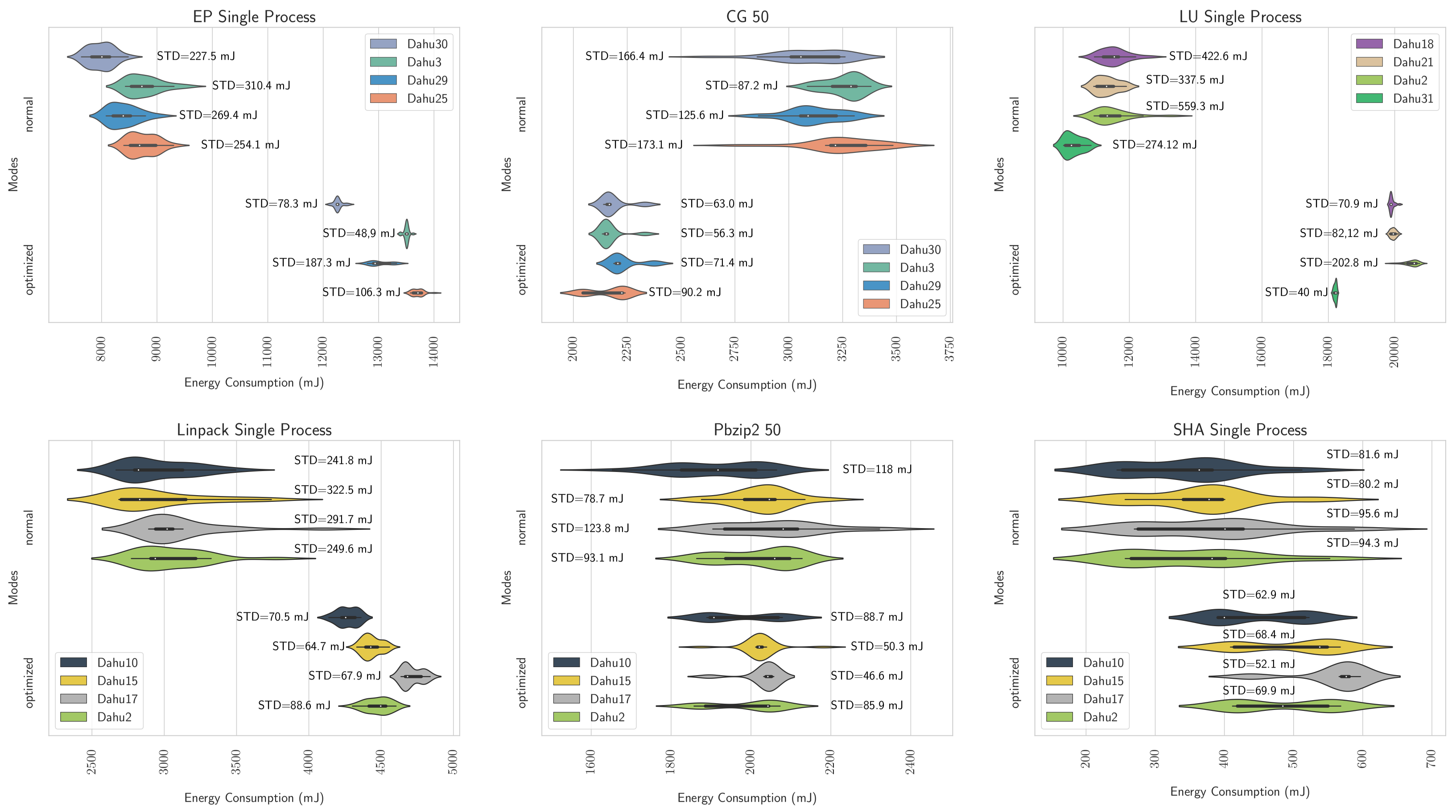


Figure 4.11: Energy variation comparison with/without applying our guidelines

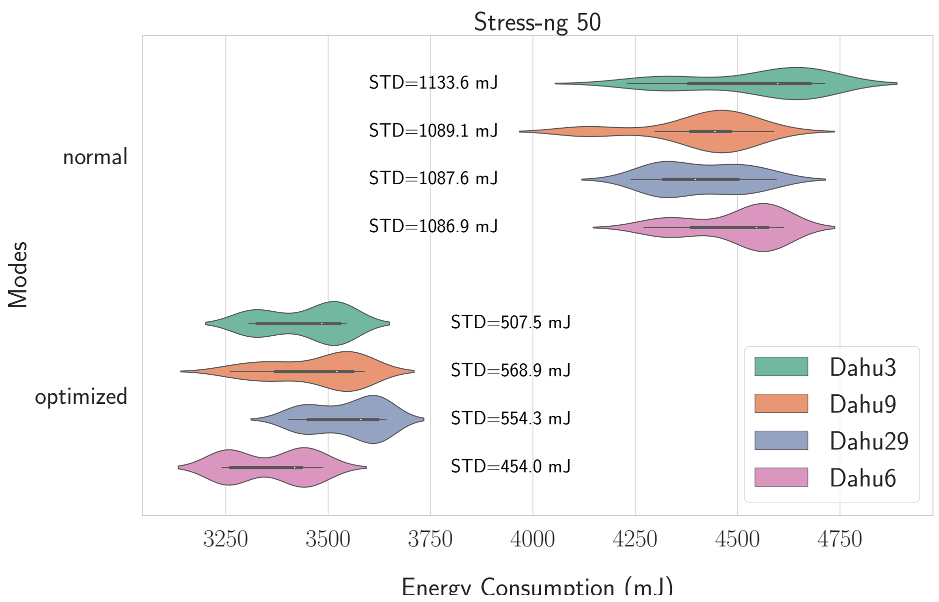


Figure 4.12: Energy variation comparison with/without applying our guidelines for STRESS-NG

4.7 Summary

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

Overall, our results are not intended to nullify the variability of the CPU, as some of this variability is related to the chip manufacturing process and its thermal behavior. The aim of our work is to be able to tame and mitigate this variability along controlled experiments. We studied some previously discussed aspects on some recent CPU, considered new factors that have not been deeply analyzed to the best of our knowledge, and constituted a set of guidelines to achieve the variability mitigating purpose. Some of these factors, like the C-states usage, can reduce the energy variation up to 500 % at low workloads, while choosing the wrong cores/PU strategy can cause up to 30× more variability.

We believe that our approach can also be used to study/discover other potential variability factors, and extend our results to alternative CPU generations/brands.

Chapter 5

Measuring and Evaluating the Energy Consumption of JVMs

Now that we have studied how to conduct robust and reproducible experiments with steady energy consumption measurements, we dive into the actions and considerations that developers can take to decrease the energy consumption of their software. For the rest of the document, we will focus on the *Java Virtual Machine*, starting with the energy efficiency of the JVM itself as an execution environment in this chapter. JVM platforms have known multiple evolutions in the last decades to enhance both the performance they exhibit and the functionalities they offer. With regards to energy consumption, few studies have investigated the energy consumption of code and data structures. However, we do miss an evaluation of the energy efficiency of existing JVM platforms and relevant configurations that can reduce the energy consumption of software running on the JVM. In this chapter, we thus assess the energy consumption of some of the most popular and supported JVM platforms using 12 Java benchmarks that explore different performance objectives. Moreover, we investigate the impact of the different JVM parameters and configurations on the energy consumption of software. Our results show that different JVM platforms can exhibit up to 100% more/less energy consumption. JVM configurations can also play a substantial role to reduce the energy consumption during the software execution.

The remainder of this chapter is organized as follows. Section 5.1 gives an overview of the chapter. Section 5.2 introduces the experimental protocol and methodology (hardware, projects, tools, and methodology) we adopted in this study. Finally section 5.3 analyzes the results of our experiments on the energy consumption of the different JVM configurations.

5.1 Overview

Software services are widely deployed to support our daily activities, being mobile or hosted in the cloud. Yet, beyond this undeniable success, the environmental impact of ICT is raising concerns and calls for solutions to reduce the energy footprint of software services [137].

Software developers often report that such solutions should come from more energy-efficient hardware components or software optimization [128, 150] but, given the complexity of modern software environments, the composition of software layers makes this sustainability objective particularly challenging.

Given this context, this chapter more specifically investigates the impact of one of these layers, the runtime environment and its settings, on the energy consumption of hosted software service. More precisely, we aim at revealing the importance of carefully selecting and configuring the *Java Virtual Machine* (JVM) to reduce software energy consumption.

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

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

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

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

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

2. Identifying and assessing the key JVM settings that can influence the energy consumption of a software service,

3. Sharing guidelines and prerequisites that will help in configuring the most energy-efficiency environment before deployment,

4. Providing a JVM benchmarking environment to evaluate the energy-efficiency of upcoming JVM distributions and their settings,

5.2 Experimental Protocol

Hardware Settings. To report on reproducible measurements, we used the cluster Dahu of the G5K platform [16] 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. Similarly to Chapter 4, our experimental protocol enforces that the software under test is the only the process executed on the node, configured with a very minimal Linux Debian 9 (4.9.0 kernel version).

Energy Measurements. We also used Intel RAPL as a physical power meter to analyze the energy consumption of the CPU package. We note that, due to CPU energy consumption variations issues (as seen in Chapter 4), we used the same node for all our experiments. To build robust experiments, every test reports on energy metrics obtained from at least 20 executions of 50 iterations per benchmarks. Having multiple iterations is useful to report on accurate measurements for very fast processes and benchmarks.

Java Virtual Machines. The latest JVM version at the time of the study was Java 15, released in September 2020, while Java 11 is the current *Long-Term Support* (LTS) version. Since Java 9, 10, 12, and 13 are no longer supported, Oracle advises developers to immediately transition to the latest version (currently Java 15), or an LTS release. Beyond HOTSPOT, one can observe that the initial JVM design leads to numerous initiatives to improve the performances of Java applications, including new hot-swapping strategies—with the *Dynamic Code Evolution Virtual Machine* (DCE VM) [152]—or alternative JIT—with GRAALVM.[1] This also includes alternative implementations, like IBM J9 JVM, which is currently distributed as part of the Eclipse foundation, and known as J9.[2] Given the wide diversity of distributions and related settings, this chapter aims to study the impact of the features implemented by available JVM distributions on the energy consumption of the hosted Java software services. To investigate this impact, we conducted a wide set of experiments on a cluster of machines, using several established Java benchmarks and JVM configurations. We considered a set of 52 JVM distributions taken from 8 different providers/packagers mostly obtained from SDKMAN!,[3] as listed in Table 5.1.

[1]https://www.graalvm.org
[2]https://www.eclipse.org/openj9
[3]https://sdkman.io/

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

Table 5.1: List of selected JVM distributions.

Depending on providers, either all the versions, major or LTS ones, are made available by SDKMAN!.

Java Benchmarks. We ran our experiments across 12 Java benchmarks we picked from Open-Benchmarking.org.[4] This includes 5 acknowledged benchmarks from the DaCapo benchmark suite v. 9.12 [23], 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 [65, 78]. 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 [124, 125], namely ALS, Dotty, Fj-kmeans, Neo4j, Philosophers, Reaction and Scrabble, which offers a diversified set of benchmarks aimed at testing JIT, GC, profilers, analyzers, and other tools. The benchmarks we picked from both suites exercise a broad range of programming paradigms, including concurrent, parallel, functional, and object-oriented programming. Table 5.2 summarizes the selected benchmarks with a short description.

5.3 Experiments and Results

5.3.1 Energy Impact of JVM Distributions

Job-oriented applications. To answer our first research question, we executed 62,400 experiments by combining the 52 JVM distributions with the 12 Java benchmarks, thus reasoning on 100 energy samples acquired for each of these combinations. Figure 5.1 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

[4]https://openbenchmarking.org

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

Table 5.2: List of selected open-source Java benchmarks taken from DACAPO and RENAISSANCE.

we consider as the baseline in this experiment. Then, we sum the ratios of the 12 benchmarks and depict them as percentages in Figure 5.1.

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

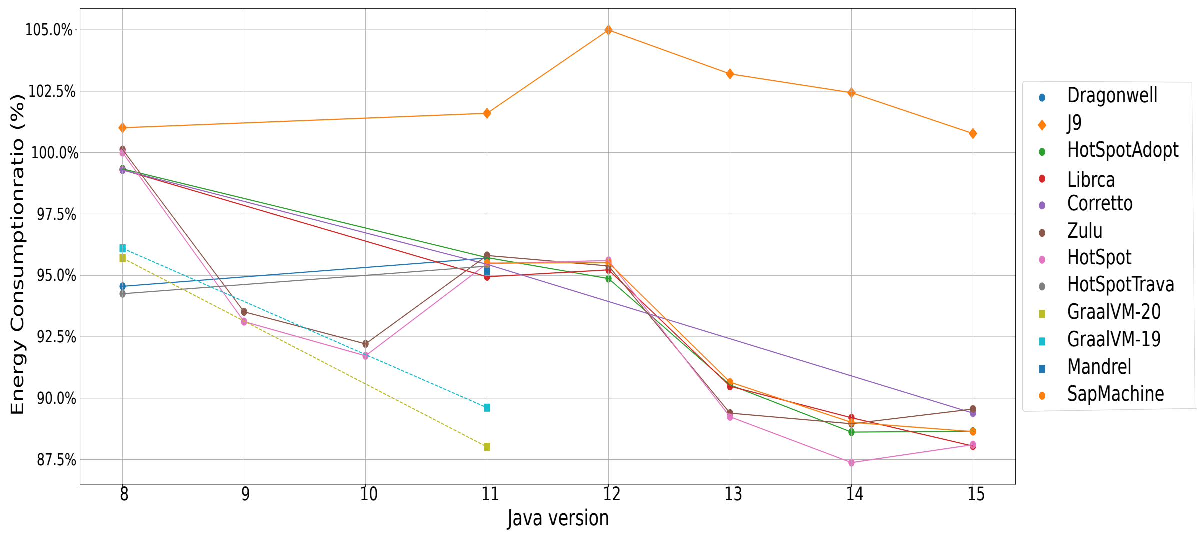


Figure 5.1: Energy consumption evolution of selected JVM distributions along versions.

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 5.1: J9, the HOTSPOT and its variants, and GRAALVM.

Then, Figure 5.2 depicts the evolution of the energy consumption of the 12 benchmarks, when executed on the HOTSPOT JVM. Figure 5.2 reports on the energy consumption evolution of individual benchmarks, using HOTSPOT-8 as the baseline. Our results show that the choice od the JVM version can impact the energy consumption of the application. However, unlike Figure 5.1, one can observe that, depending on applications, latest JVM versions can consume less energy (60% less energy for Scrabble) or more energy (25% more energy for the Neo4J). It is worth noticing that the energy consumption of some benchmarks, such as Reactors, exhibit large variations across JVM versions due to experimental features and changes that are not always kept when releasing LTS versions (version 11 here). For example, the introduction of `VarHandle` to allow low-level access to the memory order modes available in JDK 9 and works around `Unsafe Classe` that was removed from from JVM 11.[5]

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

Figure 5.3 further explores the comparison of energy efficiency of the JVM distributions per benchmark. One can observe that, depending on the benchmark's focus, the energy efficiency of JVM distributions may strongly vary. When considering individual benchmarks,

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

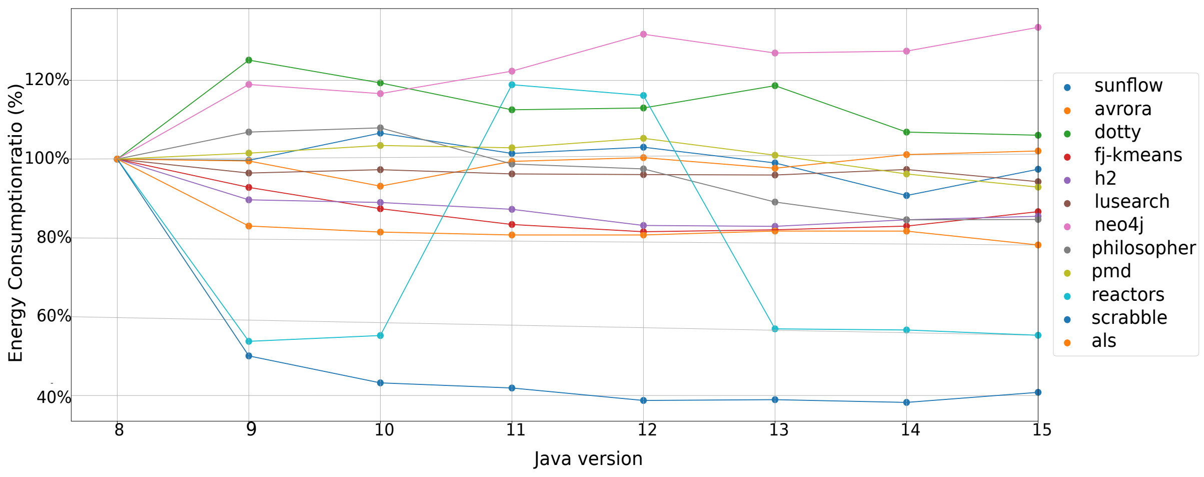


Figure 5.2: Energy consumption of the HotSpot JVM along versions.

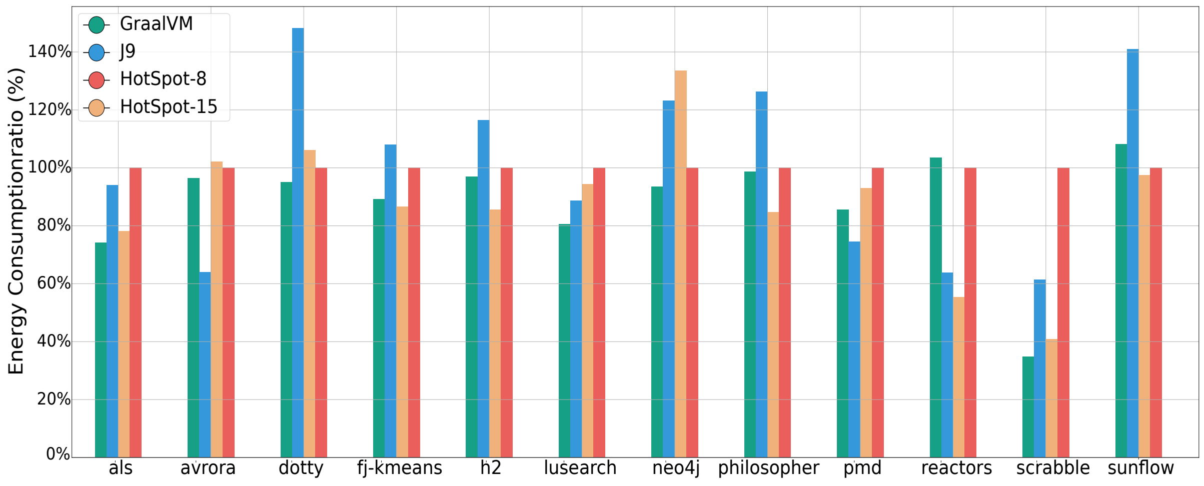


Figure 5.3: Energy consumption comparison across Java benchmarks for HotSpot, GraalVM & J9.

J9 performs the worst for at least 6 out of 12 benchmarks—*i.e.*, the worst ratio among the 4 tested distributions. Nevertheless, J9 can still exhibit a significant energy saving for some benchmarks, such as Avrora, where it consumes 38% less energy than HotSpot and others.

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

Service-oriented applications. In this part, we run the above benchmarks as services for 20 minutes, and we compare the average power and total requests processed by each of the 3 JVM

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

Table 5.3: Power per request for HOTSPOT, GRAALVM & J9.

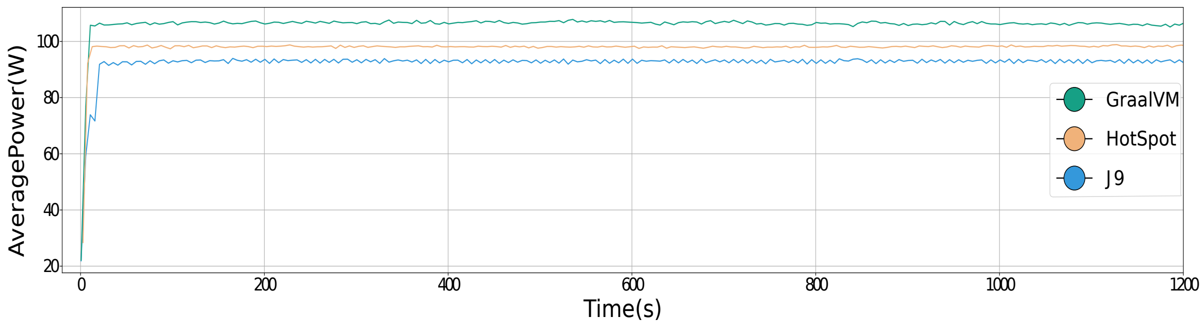


Figure 5.4: Power consumption of Scrabble-as-a-Service for HOTSPOT, GRAALVM & J9.

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 a reduction in the execution time, which incidentally results in an energy consumption reduction. Nonetheless, we can highlight 2 interesting observations for 2 benchmarks whose behaviors differ from others.

First, the analysis of the Scrabble benchmark experiments showed that JVMs can exhibit different power consumption in some scenarios. Figure 5.4 depicts the power consumed by the 3 JVM distributions for the Scrabble benchmark. One can clearly see that GRAALVM requires an average power of 109W, which is 9W higher than HOTSPOT-15 and 15W higher than J9. When it comes to the number of processed Scrabbles requests, 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 5.3. The higher power usage for GRAALVM helped in achieving a high amount of requests and a faster request execution, which was at least 40% faster on GRAALVM compare to the other two. Thus, GRAALVM was more energy efficient even if it uses more power, which confirms the results observed in Figure 5.3 for this benchmark.

The second interesting situation was observed on the Dotty benchmark. More specifically, during the first 100 seconds of the execution of the Dotty benchmark on all evaluated JVMs. At the beginning of the execution, GRAALVM has a slightly lower power consumption, is faster and consumes 10% less energy. After about 150 seconds, the power difference between the 3 JVMs is barely noticeable. One can, however, notice the effect of the JIT, as HOTSPOT-15

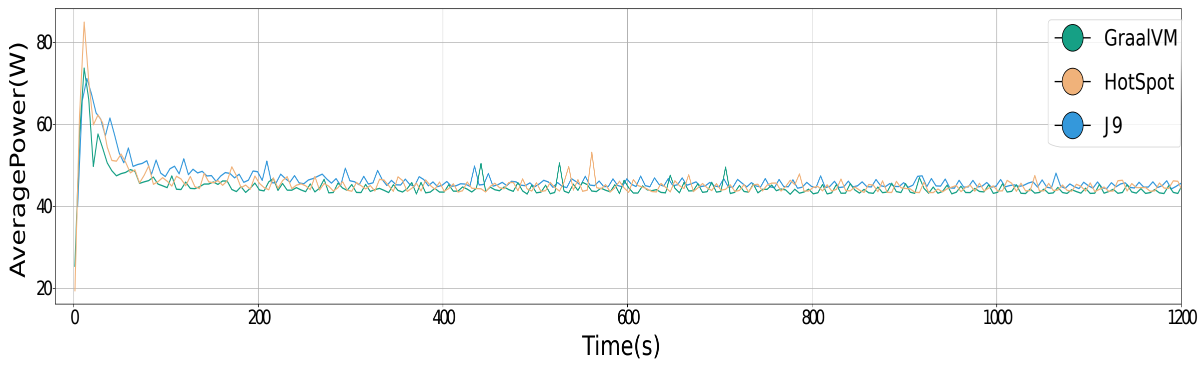


Figure 5.5: Power consumptio of Dotty-as-a-Service for HotSpot, GraalVM & J9.

takes the advantage from GraalVM and becomes more energy efficient. In total, HotSpot completes 597 jobs against 510 for GraalVM and 381 for J9, as shown in Table 5.3. HotSpot was thus the best choice for 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 on 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 5.3, thus ignoring the warm-up phase would have been misleading.

To answer RQ1, we conclude that while most of the JVM platforms are very similar, we still can cluster those JVMs in 3 classes: HotSpot, J9, and GraalVM. The choice of either of these 3 classes can have a major impact on software energy consumption, but strongly depends on the application context.

5.3.2 Energy Impact of JVM Settings

The purpose of our study is to investigate the impact of the choice of the JVM platform on the energy consumption, but also the different JVM parameters and configurations that might have a positive or negative effect, with a focus on 3 available settings: multi-threading, JIT, and GC.

Multi-threading Management

The purpose behind this phase is to investigate if JVM threads management and configuration can have a substantial impact on Java-based software energy consumption. This encompasses investigating if the management of application-level parallelism (a.k.a. threads) can exhibit a wide diversity of strategies, depending on the JVM, resulting in different energy efficiencies.

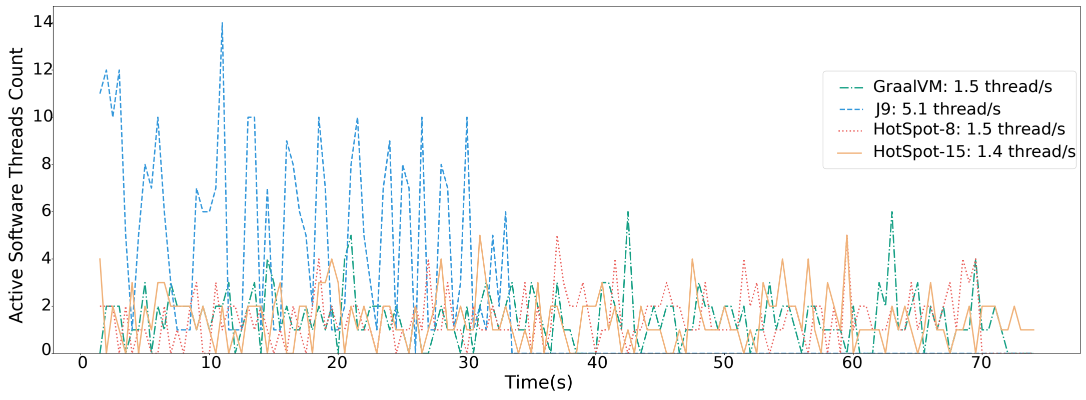
Investigating such a hypothesis requires a selection of highly-parallel CPU-intensive benchmarks, which is one of the main criteria for our benchmark selection. As no tool is capable of accurately monitoring the energy consumption at a thread level, we monitor the global power consumption and CPU utilization during the execution using RAPL for the energy part, and several Linux tools for the CPU-utilization part(Htop, CPUfreq, etc). Knowing that most of the benchmarks are multi-threaded jobs that use multiple cores, a further analysis of thread management is required to understand the results of our previous experiments. We selected the benchmarks that highlighted significant differences between the JVM distributions from Figure 5.3, namely Avrora, Reactors and Scrabble, and studied their multi-threaded behavior to optimize their energy efficiency.

Figure 5.6 delivers a closer look to the thread allocation strategies adopted by JVM. First, Figure 5.6a illustrates the active threads count evolution over time (excluding the JVM-related threads, usually 1 or 2 extra threads depending on the execution phase) for Avrora. One can notice through the figure that J9 exploits the CPU more efficiently by running much more parallel threads compared to the 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 as big for J9, while the number of soft page faults is twice as small. 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 J9 adopts a different policy for the heap allocation. It creates a non-collectable *thread local heap* (TLH) within the main heap for each active thread. The benefit of cloning a dedicated TLH is the fast memory access for separate threads, as every thread has its own heap and no deadlock can occur. However, the TLH mechanism is not always efficient, 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.

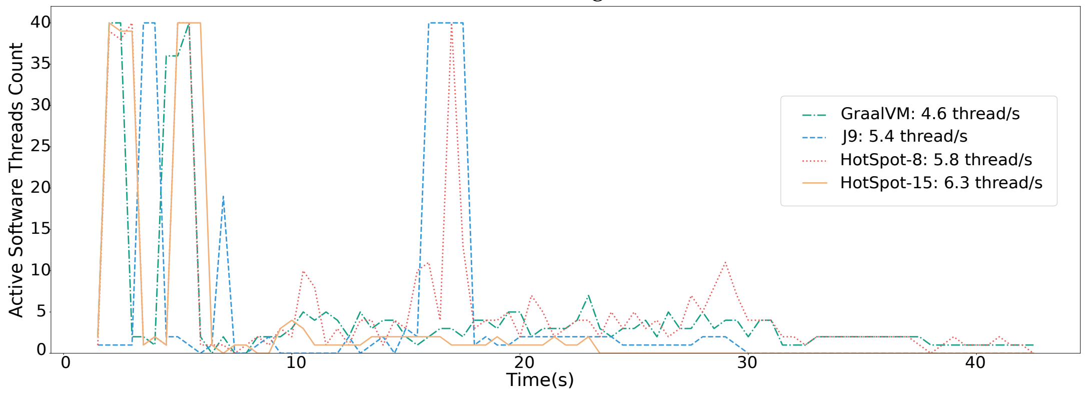
The second example in Figure 5.6b depicts the active threads evolution over time of the Reactors benchmark. One can observe that HOTSPOT-15 and J9 run faster, which confirms the results of Figure 5.3, where both JVMs consume much less energy compared to GRAALVM and HOTSPOT-8. This difference in energy consumption between benchmarks can be less likely caused by thread management for the Reactors benchmark, as HOTSPOT-8 reports a higher average of active threads.

In the case of the Scrabble benchmark illustrated in Figure 5.6c, one can see that GRAALVM executed the benchmark much faster, and with even fewer threads. With only 5.1 threads/sec, GRAALVM was able to be 50% faster than HOTSPOT-8 and J9 and consuming the least energy, while the other JVMs ran more threads per second.

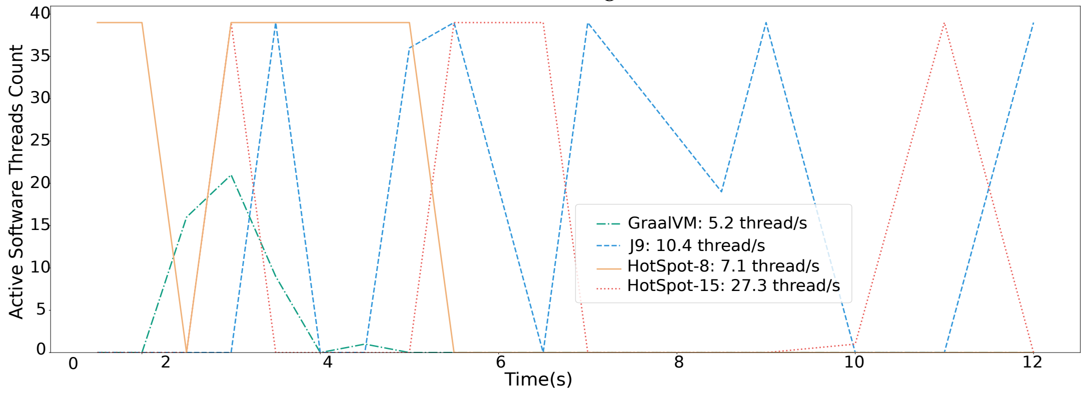
In conclusion, JVMs thread management can sometimes constitute a key factor that impacts software energy consumption. However, we suggest checking and comparing JVMs before deploying a software, especially if the target application is parallel and multi-threaded.



(a) Active threads of Avrora when using HotSpot, GraalVM, or J9



(b) Active threads of Reactors when using HotSpot, GraalVM, or J9



(c) Active threads of Scrabble when using HotSpot, GraalVM, or J9

Figure 5.6: Active threads evolution when using HotSpot, GraalVM, or J9

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).[6] The hotter the JIT, the more code optimization to be triggered. We also varied the minimum count method calls before a JIT compilation occurs (10, 50, 100), and the number of JIT instances threads (from 1 to 7). For HOTSPOT-15, we conducted experiments while disabling the tiered complication (that generates compiled versions of methods that collect profiling information about themselves), and we also varied the JIT maximum compilation level from 0 to 4. We also tried out HOTSPOT with a basic GRAALVM JIT. We note that the level 0 of JIT compilation only uses the interpreter, with no real JIT compilation. Levels 1, 2, and 3 use the C1 compiler (called client-side) with different 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 5.4 reports on the energy consumption of the experiments we conducted for most of the benchmarks and JIT configurations under study.

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

JVM	Mode	ALS		Avrora		Dotty		Fj-kmeans		H2		Neo4j		Pmd		Reactors		Scrabble		Sunflow	
GRAALVM	*Default*	2848	*p-values*	3861	*p-values*	2271	*p-values*	948	*p-values*	1959	*p-values*	3313	*p-values*	297	*p-values*	23452	*p-values*	452	*p-values*	335	*p-values*
	DisableJVMCI	3099	**0.001**	4012	**0.041**	2694	**0.001**	**934**	**0.011**	**1771**	**0.005**	5086	**0.001**	353	**0.001**	25007	**0.007**	503	**0.002**	354	0.227
	Economy	4503	**0.001**	3895	0.793	3466	**0.001**	1306	**0.002**	2560	**0.001**	9525	**0.001**	**270**	**0.001**	30317	**0.001**	649	**0.002**	392	**0.002**
J9	*Default*	3792	*p-values*	2122	*p-values*	3515	*p-values*	1271	*p-values*	2426	*p-values*	4336	*p-values*	277	*p-values*	12705	*p-values*	734	*p-values*	476	*p-values*
	Thread 1	4157	**0.001**	2121	0.875	4749	**0.001**	1297	0.097	2597	0.066	4906	**0.001**	350	**0.001**	12800	0.713	948	**0.002**	626	**0.005**
	Thread 3	3849	0.018	2105	0.713	3574	0.104	1259	0.371	2450	0.637	4477	**0.005**	294	**0.004**	12647	0.875	795	0.021	457	0.27
	Thread 7	3843	0.041	2386	0.372	3511	0.875	1259	0.25	2424	0.637	4431	0.104	273	0.372	12600	0.875	808	0.055	463	0.372
	Count 0	8461	**0.001**	2425	**0.001**	4877	**0.001**	2289	**0.002**	3212	**0.001**	10565	**0.001**	744	**0.001**	18084	**0.001**	1476	**0.002**	922	**0.001**
	Count 1	4281	**0.001**	2150	0.431	3164	**0.001**	1841	**0.002**	2546	0.431	7166	**0.001**	272	0.128	14715	**0.001**	1005	**0.002**	514	0.052
	Count 10	3980	**0.001**	2431	0.713	3771	**0.001**	1312	**0.011**	2779	**0.003**	4979	**0.001**	299	**0.001**	12000	0.104	860	**0.005**	1182	**0.001**
	Count 100	3878	**0.007**	2141	0.713	3469	0.227	1363	0.523	2513	0.128	4547	**0.001**	262	0.031	12313	0.024	768	0.16	634	**0.004**
	Cold	6788	**0.001**	2134	0.637	4855	**0.001**	1636	**0.002**	2873	**0.001**	7250	**0.001**	275	0.372	20380	**0.001**	870	**0.005**	386	**0.001**
	Warm	4594	**0.001**	2112	0.713	4253	**0.001**	1244	0.055	2521	0.128	5305	**0.001**	411	**0.001**	13726	**0.001**	913	**0.002**	336	**0.001**
	Hot	7553	**0.001**	2310	**0.001**	12749	**0.001**	1452	**0.002**	3973	**0.001**	8979	**0.001**	857	**0.001**	36534	**0.001**	1180	**0.002**	506	0.128
	VeryHot	15113	**0.001**	3300	**0.001**	18235	**0.001**	2430	**0.002**	7205	**0.001**	19359	**0.001**	793	**0.001**	38303	**0.001**	5420	**0.002**	1692	**0.001**
	Schorching	18316	**0.001**	3541	**0.001**	21686	**0.001**	2514	**0.002**	7855	**0.001**	26409	**0.014**	808	**0.001**	43929	**0.001**	5583	**0.002**	1778	**0.001**
HOTSPOT	*Default*	2997	*p-values*	4014	*p-values*	2516	*p-values*	934	*p-values*	1796	*p-values*	4787	*p-values*	323	*p-values*	11685	*p-values*	530	*p-values*	325	*p-values*
	Graal	2999	0.637	3971	0.318	2512	0.318	929	0.609	**1662**	**0.007**	4750	0.372	327	0.189	11548	0.523	537	0.701	338	0.564
	Lvl 0	491443	/	14484	/	84395	/	/	/	52344	/	356287	/	1073	/	148381	/	/	/	14559	/
	Lvl 1	/	/	**3731**	**0.001**	3302	**0.001**	1256	**0.002**	2523	**0.001**	8304	**0.001**	**222**	**0.001**	22410	**0.002**	735	**0.002**	**277**	**0.007**
	Lvl 2	3079	**0.004**	4110	0.189	3723	**0.001**	22547	**0.002**	2840	**0.001**	19058	**0.001**	226	**0.001**	40701	**0.002**	2291	**0.002**	4131	**0.001**
	Lvl 3	16375	**0.001**	7729	**0.001**	6789	**0.001**	144914	**0.002**	4139	**0.001**	44594	**0.001**	330	**0.005**	190124	**0.002**	9070	**0.002**	10449	**0.001**
	NotTired	3254	**0.001**	3901	0.189	3110	**0.001**	**912**	**0.021**	1846	0.227	**3844**	**0.001**	933	**0.001**	**11256**	**0.041**	588	**0.003**	405	**0.001**

Table 5.4: Energy consumption when tuning JIT settings on HOTSPOT, GRAALVM & J9

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

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

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

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

Globally, we conclude through these experiments that keeping the default JIT configuration was more energy efficient in 80% of our experiments and for the 3 classes of JVMs. This advocates that using the default JIT configuration 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.

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

Table 5.5: The different J9 GC policies

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

Table 5.6: The different HotSpot/GraalVM GC policies

Garbage Collection

Changing or tuning the GC strategy has been acknowledged to impact the JVM performances [79]. 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 for J9 as detailed in Table 5.5.

For HotSpot and GraalVM, we also considered many GC policies, as described in Table 5.6. Furthermore, other GC settings have also been tested for all JVM platforms, such as the *pause time*, the *number of parallel threads*, *concurrent threads* and *tenure age*.

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

Table 5.7: Energy consumption when tuning GC settings on HotSpot, GraalVM & J9

Table 5.7 summarizes the results of all the tested GC strategies with our selected benchmarks and the *p*-values of the mann-whitney test, with a null hypothesis of the energy consumption being equal to the default configuration with a 95% confidence. The *p*-values in bold show the values that are significantly different from the default configuration, where the values in green highlight the strategies that consumed significantly less energy than default. For GRAALVM, one can see that the GC default configuration is efficient in most experiments, compared to other strategies. The main noticeable impact is related to the ParallelGC and ParallelOldGC. In fact, the ParallelGC can be 13% more energy efficient in some applications with a significant *p*-value, such as Reactors, compared to default. However, the same GC strategy can cause the software to consume two times more, as for the Neo4j benchmark, due to the high communications between the GC threads, and the fragmentation of the memory.

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

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

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

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

Therefore, we noticed during our experiments that, even if using the default GC configuration ensures an overall steady and correct energy consumption, we still found other settings that reduce that energy consumption in 40% of our experiments. Tuning the GC according to the hosted app/benchmark is thus critical to reduce the energy consumption.

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

5.4 Threats to Validity

This work shares some of the threats to validity of Chapter 4 such as energy measurements errors and variability. Moreover, one major threat is benchmarks execution time, especially for some benchmarks that run fast, such as the Pmd benchmark. We thus gave a lot of attention on how long the benchmark is running for the hardware we used, and we tuned the input data workloads to execute benchmarks for at least many (from 10 to hundreds) seconds. Each experiment ran at least 30 times to compute the average consumption and the associated standard deviation, therefore reasoning over reasonable dispersion around the average.

5.5 Summary

This chapter reports on an empirical investigation of the key differences in energy consumption that some of the most famous and supported JVM platforms can exhibit, in addition to the key settings that can impact this energy consumption positively or negatively. During our experiments, we considered a total of 12 well-known and diversified-purposes Java benchmarks together with a total of 52 JVMs, including many versions of 11 different distributions. The results of our investigations showed that many JVMs share energy efficiencies and can be categorized 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 better or worse efficiency 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 only half of our experiments, with some large gains observed when using some alternative GC depending on the application characteristics.

Our main conclusions and guidelines can be thus summarized as: *i)* testing software on the 3 classes of JVM and identifying the one that consumes the least is a good practice, especially for multi-threading purposes, *ii)* while the JVM default JIT give often good energy consumption results, some settings may improve the energy consumption and could be tested, *iii)* the choice of the GC may lead to a large impact on the energy consumption in many situations, thus encouraging a careful tuning of this parameter prior to deployment.

Chapter 6

Evaluating the Impact of Java Code Refactoring on Energy

Software Energy consumption does not only concern the execution environment and configuration level. In fact, other developers' actions and choices that might impact the energy consumption of a software are at source code level. In a typical scenario of software development, a developer will have to update and maintain his/her Java software for example, after deploying it on a proper JVM and with a proper configuration. Software maintenance and evolution enclose a broad set of actions that aim to improve both functional and non-functional concerns of a software system. Among the non-functional concerns, the very famous code refactoring. In this context, the impact of code refactoring on energy consumption remains unclear. In particular, while the state of the art investigated the impact of some specific code refactorings on dedicated benchmarks, especially on mobile applications, we miss an assessment that those apply to more comprehensive and complex software. To address this threat, this chapter studies the evolution of the energy consumption of 7 open-source software developed for more than 5 years. Interestingly, the results highlight that *i)* structural code refactorings bring energy-preserving changes to the code, and *ii)* major energy variations seem to be related to functional and computational code evolution.

The remainder of this chapter is organized as follows. Section 6.1 gives an overview on what the chapter encompasses. Section 6.2 introduces the experimental protocol (hardware, projects, tools, and methodology) we adopted in this study. Section 6.3 analyzes several experiments we conducted to mine the code refactorings and evaluate their impact on the energy consumption, as well as the results we observed during these experiments.

6.1 Overview

Software energy consumption has gained a substantial significance in the last decade, both for research and industrial contexts [30, 44, 122, 128, 146]. Hence, many researchers and practitioners started caring about the energy efficiency of software, beyond performance and hardware concerns [35, 91, 88, 120]. Being integrated into mobile or cloud environments, software systems are trying to minimize their resource consumption to reduce battery consumption or operational cost.

In this context, the impact of software development techniques on energy consumption has been explored by the state of the art, including code compilation, static code analysis and code refactoring, which is the focus of this chapter. Source code refactorings can be described as the application of acknowledged rules to improve one or many aspects of a software system, such as its clarity, maintenance, code smells, without impacting its functional behavior [3, 66].

Yet, code refactoring has also been considered as a mean to improve the performance and/or energy efficiency in a more or less automated way [11, 25, 35, 36, 48, 99]. The large majority of the literature studies that has been published in this domain—especially for mobile application [11, 48, 80, 113]—based their study on a predefined set of refactoring rules, design patterns, or code smells. In most of these studies, the authors measure and analyze the effect of atomic code changes on the total energy efficiency of the software under study, before concluding on their effect. While this process may deliver interesting insights on the impact of specific code refactorings on the energy consumption of a code snippet, there is still no guarantee that the identified code refactorings are frequently applied during the lifespan of a software system.

In this chapter, we thus consider an alternative approach to study the impact of code refactorings on the energy efficiency of legacy software systems. We focus on acknowledged refactoring rules mostly issued from Fowler's book [46], which are mostly structure-oriented rules (such as Extract Method) dealing with code architecture and organization for server-side applications rather than implementation and computation changes (such as Substitue Algorithm). Instead of selecting a set of code refactorings *a priori* and evaluating them against some dedicated benchmarks, we extract these code refactorings from established open-source projects. More specifically, we mine the history of code refactorings that have been applied to these projects in the past, and we measure the impact of the commits that include acknowledged code refactorings on the overall energy consumption. This approach aims to detect the code refactorings that have been broadly applied, and their observable impact on energy efficiency in practice. By doing so, we believe that mined code refactorings are most likely to reflect an effective impact of code refactoring on energy consumption, compared to the study of a fixed set of refactoring candidates. This study, therefore, aims to answer the following research questions:

RQ 1: How does the energy consumption of software evolve over time?

RQ 2: How do code refactorings contribute to the evolution of software energy consumption?

The chapter comes with a set of contributions that can be summarized as:

1. Proposing a new empirical approach to study the impact of structure-oriented code refactorings on the energy consumption of software systems,

2. Investigating the contribution of code refactorings to the global evolution of software energy consumption,

3. Providing a detailed description of the most applied code refactorings and their impact on energy consumption,

4. Validating the code refactoring effects on energy consumption through statistical tests and micro-benchmarking.

6.2 Experimental Protocol

This section describes our detailed experimental environment, encompassing the hardware configuration, the studied projects/benchmarks and a detailed description of our experimental methodology. We note that benchmarking protocol and environment configurations similar to Chapter 4 were considered for experiments accuracy.

6.2.1 Hardware Environment

For all of our experiments, we used a `Core i7` machine (i7-6600U CPU @ 2.60GHz) with a total of 4 processing units to measure the energy consumption and mine the refactoring rules from the projects under study. The machine ran a 18.04.4 LTS Ubuntu, with a `4.15.0-88-generic` Linux kernel. We also used OPENJDK, version 1.8.0_242, to run most of our Java experiments—*i.e.*, run both old and recent versions—except for the `OkHttp` project where we had to use OPENJDK, version 11.0.6. By using the same machine to conduct all the experiments, we guarantee the least energy consumption variation and a controlled impact of the hardware configuration.

6.2.2 Projects Under Study

Regarding the subjects of our study, our main criterion was to select established projects with a considerable commit history, that have been existing for years, and with an active community. This study exclusively focuses on Java projects to limit the search space and unify our experimental setup, but also because code refactorings may differ from a language/paradigm to another. We then tried to diversify our dataset by considering projects that cover a large spectrum of features and operations including, JSON and XML conversions, HTTP client, graph processing, data collections, etc. Because of the longitudinal nature of our study, we

Project	Description	# commits	1st commit
OkHttp	Java HTTP client	4,684	05-2011
JGraphT	Graph objects and algorithms provider	3,158	07-2003
XStream	XML $\leftrightarrow$ Java objects serialization	2,736	10-2003
JFlex	Java lexical analyzer generator	1,741	02-2003
Gson	JSON $\leftrightarrow$ Java objects serialization	1,485	08-2008
Eclipse-Collections	Eclipse Java collections	1,374	12-2015
Google-Http	Google HTTP client library for Java	868	05-2011

Table 6.1: List of selected open-source projects

considered projects that have a stable interface, and in which the main functions are unambiguously identified, so we can run the same measurements across different generations and versions of the studied projects.

Based on the above criteria, Table 6.1 summarizes the projects that we considered for this study, along with the number of commits at the time that this chapter was written, and the date of the first commit. Established projects with a higher number of commits increase the chances to mine a representative set of commits including code refactorings. All the projects we selected have been hosted on GitHub since at least 2015. We note that the Git creation date only gives an overview of how long the project has been on GitHub and is different from the project creation date. Some projects, such as Gson, exist on GitHub since March 2015, but we still can checkout commits from 2003.

6.2.3 Methodology and Tools

Our experimental methodology is a process that includes extraction, evaluation, and validation steps. Figure 6.2 depicts the main steps we followed to analyze each selected project. We start our process by cloning the public repository of the project from GitHub. Then, for each commit, we mine the code refactorings of the project using the RefactoringMiner tool and we summarize them into a JSON file. RefactoringMiner is an open-source research project [140, 139] that analyses a project commit by commit and extracts the type and count of refactorings for each commit in a JSON format. It helps in detecting and visualizing 55 different types of refactoring in its version 2.0, which is the version we used in this study.[1]

Once we extract the code refactorings that have been applied per commit on the master branch, we select the commits to be reproduced to measure their energy consumption. The selection method takes into account the refactorings count and types in each commit. We consider commits with a threshold of 20 refactorings so we can expect a significant impact of the refactorings on the energy consumption. Figure 6.1 depicts the *cumulative distribution function* (CDF) that shows the frequency of commits per refactoring count (commits with more

[1] https://github.com/tsantalis/RefactoringMiner

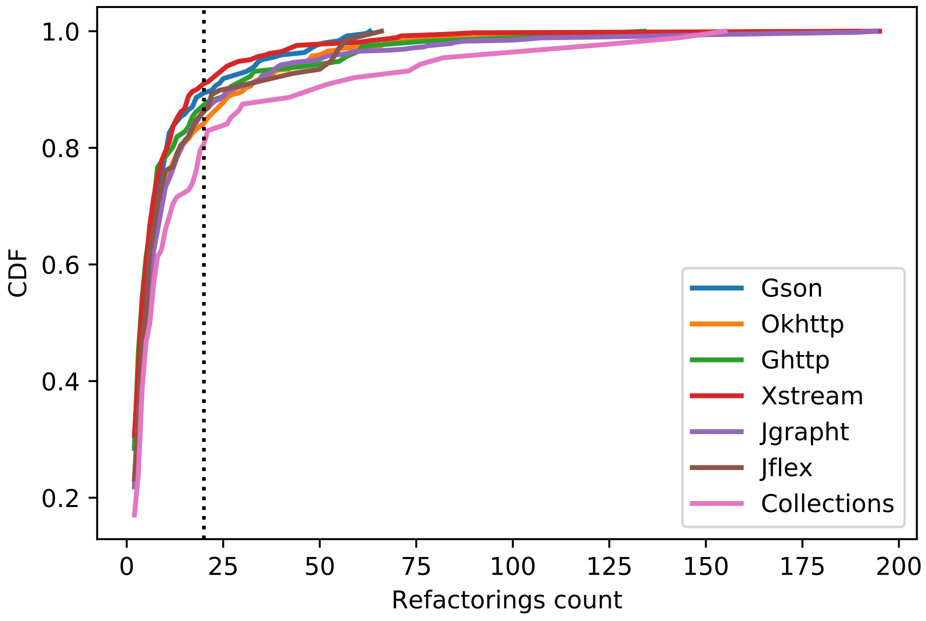


Figure 6.1: CDF of code refactorings per commit.

than 200 refactorings have been omitted for clarity). For most of the studied projects, one can see that 20% of the commits have more than 20 refactorings. Since the commits that contain only one type of refactoring are very rare, we also consider commits with a mix of code refactorings and deduce the impact of each refactoring rule *a posteriori*.

Then, we rebuild the project *Java archive* (JAR) for each of the previously selected commits to be ready for the test/run phase. To be able to run and evaluate the compiled JAR, we need to provide a task to execute for each project. We cannot trust running the tests provided within projects as they can substantially change from a commit to another and might include/exclude functionalities that appear/disappear between commits, which does not constitute a fair comparison criterion. Instead, we wrote our own JMH benchmarks for each project, which is a "*Java harness for building, running, and analyzing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM*".[2] The purpose of each benchmark is to test the main functionality of each project to ensure the same measurement conditions for all commits. Hence, through JMH benchmarking, we can deliver—for each project—experiments to compare the energy consumption of commits, while testing the main functionalities of the project. The main test functionality for Gson and XStream is JSON and XML to Java objects serialization and deserialization, respectively. For both OkHttp and Google-Http projects, we consider the core HTTP verbs (`GET`, `POST`, `DELETE`) with a local server to eliminate any network bias. For JGraphT, we consider the operations of graph creation, shortest path computation, max-flow computation, and discarding random edges. We also tested JFlex with lexical

[2]https://openjdk.java.net/projects/code-tools/jmh/

analyzer generation, and Eclipse-Collections with the core operations on the different mutable and immutable collections (lists, maps, sets), inspired from Hasan et al.'s and Pinto et al.'s studies [52, 122]. Using JMH for writing benchmarks has many advantages, such as the easy management of run and warm-up iterations, and the prevention of dead code removal from the JIT using the concept of *blackhole* [129].

Once the JMH benchmark was written, we compute the coverage of the project by the benchmark using Jacoco (https://www.eclemma.org/jacoco). The purpose is not to cover all of the project classes and methods, as we only want to test the main functionality of the project. However, the coverage computation allows us to save all the classes and methods that are covered by our benchmark. Thus, only the commits with refactoring on these classes (given by RefactoringMiner) and methods are considered for the evaluation. Of course, this operation requires applying more checks to ensure that the changes of the commit `x` ***are limited to the extracted refactorings and nothing else susceptible to affect the performance or the energy consumption***. Hence, this step ensures that the selected commits only contain refactorings that are being stressed by our benchmark.

The next step is to run the benchmarks for each of the JAR files compiled from relevant commits. To highlight the effect that code refactorings may have on energy consumption, we build and run the commit `x` that includes the code refactorings, but also the commit `x-1` on the main branch, so we can compare the energy consumption and infer the impact of refactorings.

The percentage of reproduced commits, which designates the ratio of successfully built and ran commits in regards to the total count of selected commits (Gson: 95%, XStream: 80%, OkHttp V3 and V4: 90%, Google-Http: 15%, JGraphT: 25%, JFlex: 40%, Eclipse-Collections: 50%). Most of the unsuccessful projects' rebuilds are due to deprecated and invalid references.

During the execution of the experiments, we use Intel RAPL to acquire the global energy consumption. We thus evaluate the energy consumption of every commit `x` and we compare it to its `x-1` commit. We run every JMH benchmark for multiple iterations on a fixed amount of time, and we extract between 100 and 1,000 energy measurements depending on the duration of each iteration. Thus, different commits can run a different amount of iterations within the time allowed to the JMH benchmark execution. This is why we consider the energy consumption of iterations rather than the whole benchmark, in order to have a correct estimation of the energy consumption for that commit. Then, similarly to Chapter 4, we use the bootstrap method [40] to randomly build 100 subsets from the main set of measurements, and we compute the mean and standard deviation of these subsets. We end-up with 100 measures of averages and we use the median of these values for better accuracy and less bias.

The checked results are then used to build global statistics of the most efficient refactoring rules across the selected commits of all projects. This additional check of commits consists of applying a more detailed `git diff` analysis on the results of the previous step to verify every single occurrence of the detected refactorings, that they contain no other changes that may affect the energy efficiency. Another check consists of an extra micro-benchmarking phase,

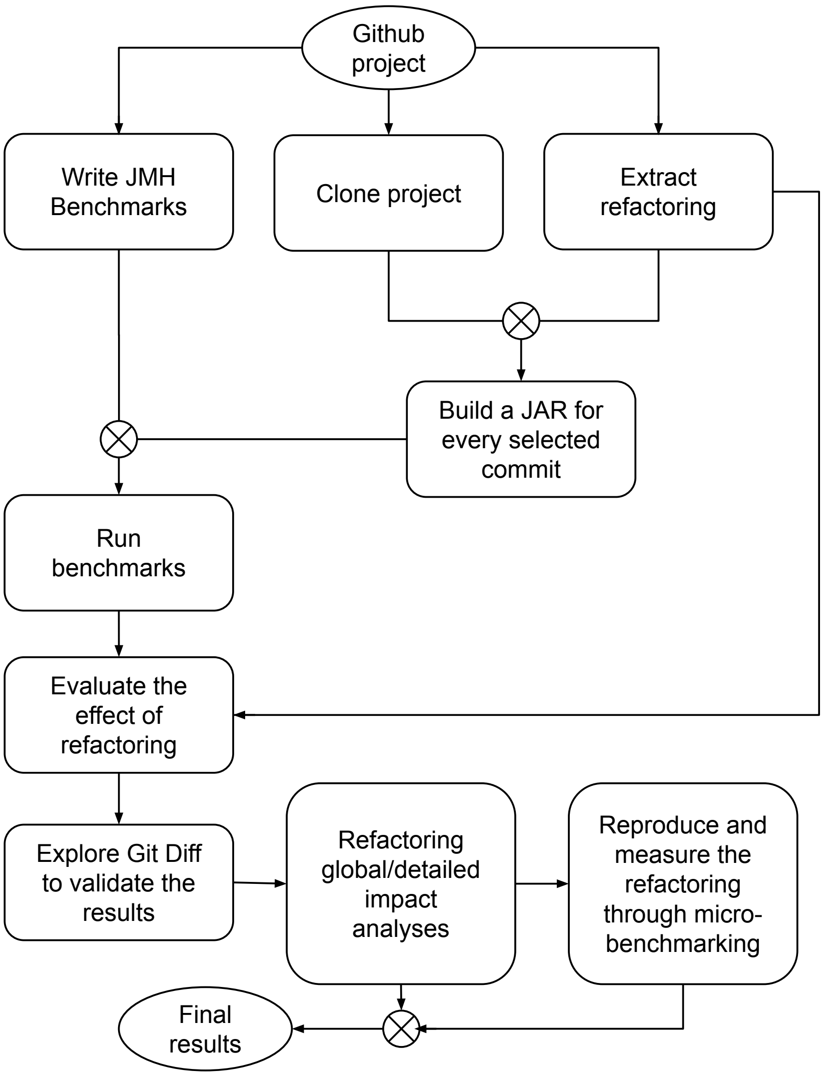

Github project
Write JMH Benchmarks
Clone project
Extract refactoring
Build a JAR for every selected commit
Run benchmarks
Evaluate the effect of refactoring
Explore Git Diff to validate the results
Refactoring global/detailed impact analyses
Reproduce and measure the refactoring through micro-benchmarking
Final results

Figure 6.2: Methodology of refactoring analysis

where we prepare and execute the extracted refactorings to confirm and validate the effect they could have on the energy efficiency of the project/software. We also applied the Wilcoxon rank sum test (or Student test when possible) to check the statistical significance of the registered difference in the energy consumption between the commit `x` and the commit `x-1`, with a null hypothesis of the energy consumption of the commit `x` and `x-1` being equal with a 5% certainty.

6.3 Refactoring Impact Analysis

In this section, we aim at answering our research questions with a clear conclusion on whether refactoring has a substantial impact on the evolution of software energy consumption over time. We, therefore, conducted a set of experiments and validations to investigate the effect of structural refactoring on the evolution of software energy consumption.

6.3.1 Software Energy Consumption Evolution

The first step is to investigate the evolution of software energy consumption over time. Figure 6.3 depicts the evolution of energy consumption for the projects Google-Http, XStream, JGraphT, and Eclipse Collections, for which we run the main releases and report on the energy consumption measured over time, by focusing on the main functions stressed by our JMH benchmarks.

Except for JGraphT, one can observe that energy consumption tends to decrease over time for most of the projects. One can mention a 10% decrease in 12 months for the Google-Http project (cf. Figure 6.3a), a 10% decrease in 4 years for the Eclipse Collections project (cf. Figure 6.3c), and a very substantial decrease of 50% in 6 years for the XStream project (cf. Figure 6.3d).

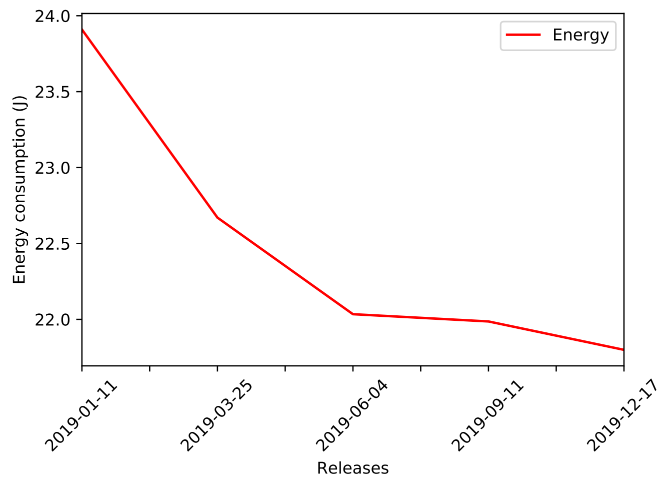
Then, to have a more concrete look on the evolution of energy consumption per commit, we select the Gson project to reproduce the evolution of its energy consumption along the full commit history. Given the large number of involved commits, we consider the full set of commits of the Gson project (12 years) with a span of 25—*i.e.*, we build, run, and measure the energy consumption every 25^{th} commits. Figure 6.4 depicts the evolution of energy consumption for the Gson project with a total of 57 successfully reproduced commits, out of 60. The line plot validates and confirms the results shown in Figure 6.3. Most notably, one can observe a reduction of 82% from the highest to the lowest consumption commit within 12 years of the project's lifespan—*i.e.*, the energy consumption became 5 times lower. One can also see a more sudden energy consumption reduction between commits 600 and 900. This requires further investigation in the future.

To answer RQ1, we highlighted that software energy consumption can **evolve drastically** over time. For the analyzed target systems, in spite of fluctuations, the energy consumption has decreased non-negligibly for 4 systems and grown for one.

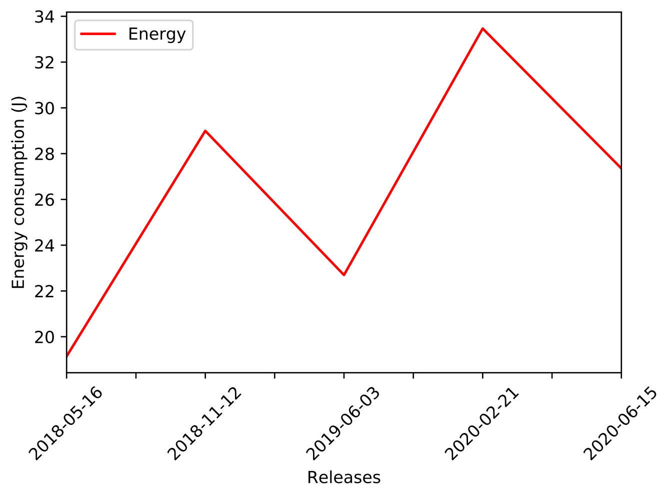
Given the previous results reported by the literature, the remainder of this chapter aims to closely study and assess the impact of code refactoring on such observed evolutions.

6.3.2 Refactoring Rules Impact

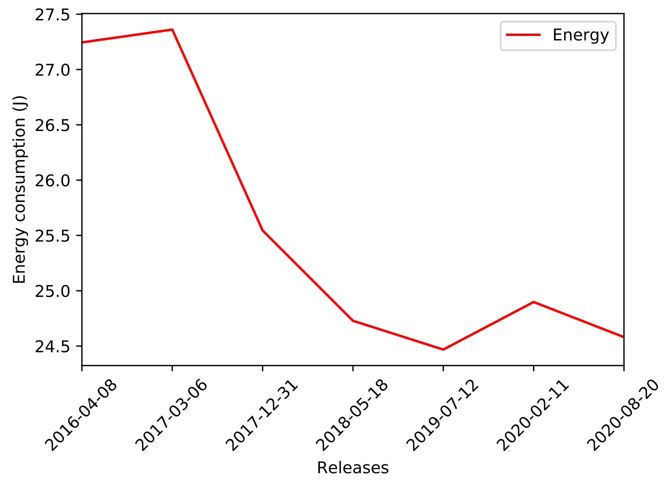
To dive into the effective impact that code refactoring may have on software energy consumption, we further tracked and analyzed the evolution of the energy consumption on commits where code refactorings were detected. Thus, in our study, we consider the full commit history of 7 open-source projects, and we analyze the impact on energy consumption of commits including code refactorings, as described in Section 6.2.



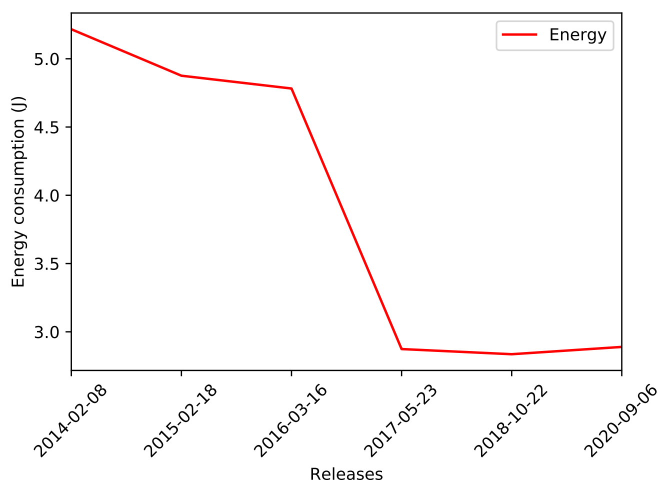
(a) Google-Http energy consumption over 11 months



(b) JGraphT energy consumption over 2 years



(c) Eclipse Collections energy consumption over 4 years



(d) XStream energy consumption over 6 years

Figure 6.3: Energy consumption evolution of Google-Http, XStream, JGrapht, and Eclipse Collections.

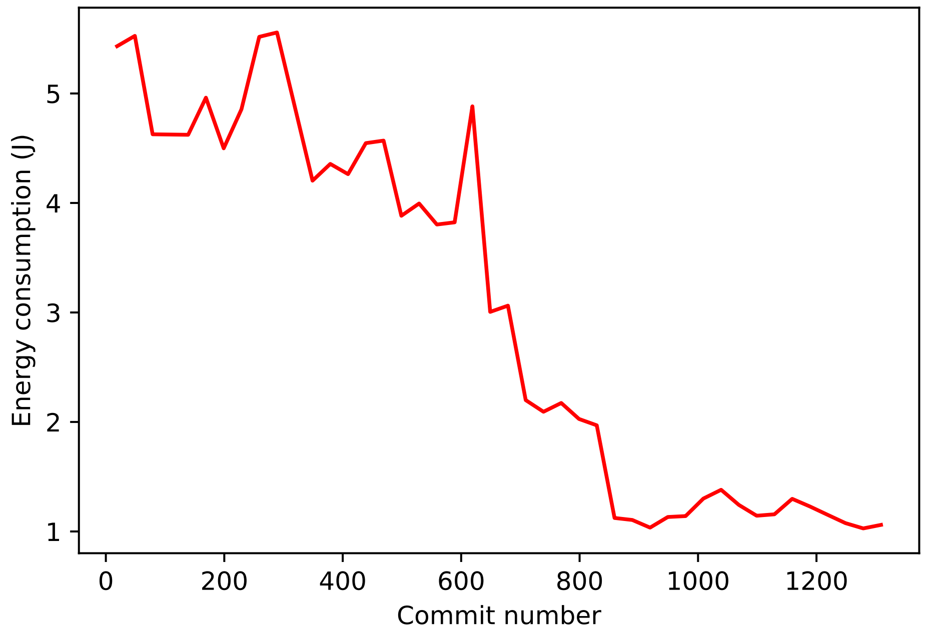


Figure 6.4: Gson energy consumption across all commits.

Once we select commits with code refactorings and rebuild them, we run the JMH benchmarks that have been prepared for each project to compare the energy consumption of a commit `x` that includes the refactorings and the previous commit `x-1` on the master branch.

Then, we report on global statistics from the raw measurements we obtained from each project, thus establishing a summary of the most used code refactorings and their impact.

Global Code Refactoring Statistics

The purpose of this step is to highlight the most used/impactful code refactorings. While it is easy to identify the most used code refactorings by counting the number of occurrences of each refactoring rule and the commits they appear in, there is no consensus on how to measure the effective impact of code refactorings on energy consumption, if any. The large majority of commits comes with a set of code refactorings of many types, and even if these refactorings can impact the energy consumption, there is no trivial way to isolate such an impact for each type of refactoring. Thus, we consider 3 indicators to capture the energy impact of refactoring. The first indicator, *Impact in Commits* (IC), is the ratio between the number of commits where the refactoring had a positive or negative impact—*i.e.*, the commit `x` containing this refactoring consume more or less energy than the previous commit `x-1`—and the total number of commits containing this refactoring. Equation 6.1 therefore computes IC for a rule $r \in R$ by exploring all the commit history C of a given project:

$$\mathrm{IC}(r) = \frac{\sum_{c \in C} count\_positive\_negative(c,r)}{\sum_{c \in C} count(c,r)} \tag{6.1}$$

This indicator can be then enhanced by taking into account the occurrences—or weights—of each refactoring rule in a commit. In other words, considering the refactoring weight consists of using the number of occurrences of each refactoring type within a commit rather than only counting the commit as 1 if it contains at least a refactoring.

$$\mathrm{WIC}(r) = \frac{\sum_{c \in C} count\_positive\_negative(c,w_r)}{\sum_{c \in C} count(c,w_r)} \tag{6.2}$$

Nevertheless, this indicator is not enough to evaluate the energy impact of refactoring. Indeed, including the weight of refactorings in commits supposes that all refactorings impact energy consumption equally, which may not be true, as we assume that the occurrence of a refactoring r_1 can have a bigger impact than many occurrences of a refactoring r_2.

The 2nd and 3rd indicators are $\delta\%$ and $\delta|\%|$ that indicate the mean of the energy consumption of every commit `x` containing the refactoring minus the energy consumption of commits `x-1`, and the mean of the absolute value of the energy consumption of every commit `x` containing the refactoring minus the energy consumption of commits `x-1`, respectively, $\|C_r\|$ being the commits in the commit history C where refactoring r occurred.

$$\delta\%(r) = \frac{\sum_{x=1}^{C_r}(E_x - E_{x-1})}{\|C_r\|} \tag{6.3}$$

$$\delta|\%|(r) = \frac{\sum_{x=1}^{C_r}|E_x - E_{x-1}|}{\|C_r\|} \tag{6.4}$$

where E_x and E_{x-1} represent the mean energy consumption of the commit `x` that includes at least the refactoring r, and the energy consumption of the commit `x-1`, respectively. These indicators are complementary to reflect the impact of the code refactorings on the energy consumption. Therefore, we consider an aggregate indicator that combines the previous indicators to capture the energy impact of refactorings across commits. This indicator, named *Refactoring Impact* (RI) builds on the previous indicators: the higher WIC and $\delta|\%|$, the more impactful the refactoring r is. However, if the difference $\delta|\%|-\delta\%$ is high, it means that the refactoring r has an unpredictable effect on the energy consumption and may affect the energy consumption positively or negatively. This is a negative effect and could mean that the refactoring does not have any impact at all. On the other hand, the more commits we have with the refactoring r, the more certain we are of the effect that it could have. Thus, we use the exponential function in Equation 6.5 so the denominator cannot be null.

$$\mathrm{RI}(r) = \frac{\mathrm{WIC}(r) \times \delta|\%|(r)}{e^{\delta|\%|(r) - \delta\%(r)}} \times \|C_r\| \tag{6.5}$$

Table 6.2 shows the computed indicators for a total of 25 mined refactoring rules. We note that the commits that could not be reproduced and those where the refactorings are parts of classes that are not tested by our benchmark have already been discarded and not displayed in Table 6.2. Before analyzing the results we excluded all the code refactorings with a low number of occurrences and/or commits (less than 20 CountxCommits). For example, code refactorings that occurred only a couple of times and/or only in one or two commits cannot be faithfully studied due to insufficient data. Then, we highlight (in Cyan) the refactoring rules that have the best values for the previous indicators, which are very likely the refactorings with the most impact on energy consumption. The 4 refactoring rules with the most number of occurrences and commits, with a minimal IC of 30%, are "add method annotation", "rename parameter", "add class annotation", and "move class". These refactoring rules are also those that exhibit the highest RI, and thus, are most likely to be the most impactful on energy consumption. However, we still have to assess that these refactoring rules have an effective impact on the evolution of energy consumption. Thus, we conducted a more detailed study on the commits with the highest impact to validate the effect of code refactorings on energy consumption.

Diving Into the Most Impactful Commits

With the most impactful commits, we refer to commits where we observed the most substantial energy differences between the commits `x` and commit `x-1`. To select these commits, we fix a threshold of 5% in energy consumption difference. This threshold was fixed based on the CPU energy consumption variation observations in Chapter 4 and the standard deviation of the many executions we ran on the same test, which is often around 4% to 5%. A total of 7 commits have been retrieved from the projects Gson, JFlex, Eclipse-Collections and JGraphT (no other refactoring commit with a minimal impact of 5% has been observed among the other projects). We note that our experimental setup would highlight any effect that these refactoring could have caused on energy consumption. Indeed, the execution of a JMH code, which uses the compiled JAR for the commit `x`, is composed of numerous warmup and standard iterations. Each iteration itself consists of running the benchmark many thousands of times for several seconds, so the effect of the difference between the commits `x` and `x-1` could be noticed, if any.

Table 6.3 reports on the most impactful commits including code refactorings. For each commit, we can see the type and number of refactorings extracted using REFACTORING-MINER [140, 139], the measured energy consumption difference, a short description of the refactoring-related changes that have been observed within the commits, and the computed p-value of the Wilcoxon test.

Refactoring	**Count**	**CountxCommits**	**IC**	**WIC**	$\delta\%(r)$	$\delta\|\%\|(r)$	**RI**
add method annotation	10120	**80960**	**30.77%**	**43.41%**	**1.13%**	**2.14%**	**7.34**
change variable type	101	**606**	16.67%	14.95%	0.24%	1.32%	1.17
rename parameter	45	**180**	**33.33%**	**71.69%**	-0.07%	**1.82%**	**5.12**
change parameter type	42	**168**	11.76%	17.07%	-0.03%	1.20%	0.81
change attribute type	26	**130**	16.67%	9.39%	0.12%	1.35%	0.63
add class annotation	63	**216**	**33.33%**	**63.53%**	**1.30%**	**2.20%**	**2.77**
move class	40	**120**	**30.00%**	**54.28%**	**0.77%**	**2.21%**	**3.55**
change return type	28	**112**	14.81%	19.93%	0.14%	1.11%	0.88
move method	33	**99**	21.43%	19.10%	0.59%	1.76%	1.00
rename variable	21	**84**	25.00%	18.24%	0.46%	1.44%	1.04
move attribute	18	**54**	25.00%	18.81%	-0.07%	1.92%	1.06
extract method	37	**37**	20.00%	71.87%	0.08%	1.24%	0.88
pull up method	32	**32**	33.33%	38.90%	0.03%	1.97%	0.75
rename class	6	**24**	25.00%	13.71%	**1.14%**	**1.51%**	0.82
add attribute annotation	8	16	20.00%	15.12%	0.64%	1.14%	0.34
rename attribute	5	15	30.00%	8.77%	0.55%	1.62%	0.42
add parameter	6	12	16.67%	6.55%	0.82%	1.47%	0.19
merge parameter	6	6	100.00%	100.00%	6.00%	6.00%	6.00
extract class	2	4	33.33%	11.14%	0.72%	2.62%	0.57
extract variable	3	3	11.11%	10.52%	0.49%	0.91%	0.10
remove method annotation	1	1	11.11%	0.77%	0.71%	1.40%	0.01
rename method	1	1	11.11%	2.20%	0.32%	1.10%	0.02
modify method annotation	1	1	33.33%	7.99%	2.50%	2.50%	0.20
move & rename method	1	1	20.00%	13.17%	-0.32%	2.32%	0.30
merge attribute	1	1	100.00%	100.00%	6.00%	6.00%	6.00

Table 6.2: The observed impact of mined refactoring rules

Project	Commit ID	EC diff	Refactoring	Count	Git diff	*p*-value
Gson	`#82771f`	5.5%	add method annotation	23	Adding `@SuppressWarnings("unused")` and `@SuppressWarnings("unchecked")` to methods, classes and variables that appear in the call trace of the JMH code with no other changes that might impact the energy consumption.	0.018
			add class annotation	3		
			modify method annotation	1		
			add attribute annotation	1		
	`#45bf2d`	6.8%	add method annotation	3	Adding `@SuppressWarnings("unchecked")` to methods and moving classes (project reorganization) that appear in the call trace of the JMH code.	0.000
			move class	30		
JGraphT	`#033164`	6%	merge attribute	1	Some code restructuring, reorganization and class movement that that appear in the call trace of the JMH code. No other changes suspected of impacting the energy consumption were detected	0.056
			change parameter type	1		
			rename parameter	9		
			move method	22		
			rename class	1		
			extract class	1		
			move attribute	15		
			move class	8		
			merge parameter	6		
			change variable type	19		
			change attribute type	1		
	`#f1074b`	5%	add method annotation	1	Adding `@Override` annotation and the renaming of some attributes/parameters. However these changes does not appear in the call trace of the JMH code.	0.2
			add class annotation	60		
			rename class	2		
			rename attribute	1		
			change variable type	16		
			rename parameter	4		
JFlex	`#b34361`	5%	add method annotation	53	Adding `@override` annotation to methods that appear in the call trace of the JMH code with no other changes that might impact the energy consumption.	0.054
			move & rename method	1		
			rename class	1		
Eclipse Collections	`#b9dfbc`	6%	add method annotation	9944	Adding `@override` annotation to methods that appear in the call trace of the JMH code with no other changes.	0.4
	`#298b7a`	5%	add method annotation	73	Adding `@override` annotation to methods that appear in the call trace of the JMH code, but too many changes unrelated to refactoring were found.	0.01

Table 6.3: A deeper look into the most impactful commits

First, the commit ID is the first 6 digits of the git hash that can be used to access the commit and reproduce our experiments/results. The *energy consumption* (EC) difference represents the percentage of differences between the average measure of commits `x` and `x-1` (after applying the bootstrapping as we compute the average of multiple subsets built from the main set of values). The next 2 columns contain the extraction results of the RefactoringMiner tool. They include the type and count of each refactoring the tool was able to extract. We notice that the rules that we identified as most impactful in the previous phase (`add method annotation`, `rename parameter`, `add class annotation`, and `move class`) are—most of the time—part of the extracted rules in theses commits that have shown the highest differences in energy consumption, with `add annotation` and `move class` being the most common. Sometimes, they are the only detected code refactorings, that we could suspect to be responsible for the energy consumption variation, as in commit `#b9dfbc` of `Eclipse Collections`.

We apply 3 different validation measures to confirm whether the impact is effectively caused by the refactoring. The first validation is through detailed `git diff` checks of the 7 selected commits to assess that the refactorings have been faithfully applied. We remind that we have already made sure that these refactorings only concern classes and methods that are being stressed by the JMH benchmarks, and do not contain other changes that can be responsible for the energy consumption difference. For example, we do not suspect adding some code documentation to alter the energy consumption, yet we do suspect changing a data structure, a loop, or a code snippet to do so.

In the second validation step, we conduct a statistical validation through Wilcoxon rank sum test (as Student test could not be applied due to variables not following a Gaussian distribution) to compare the commits `x` and `x-1` averages. With a risk of 5%, we reject the null hypothesis of the means of the executions of commits `x` and `x-1` being equal. For the p-value commit `#f1074b` being higher than 0.05, we cannot reject the possibility that the average is equal in both commits. The same goes for the commits `#033164`, `#b34361`, `#b9dfbc` where we cannot accept that the means of the commits `x` and `x-1` are statistically different.

The remaining commits—being `#827717`, `#45bf2d`, and `#298b7a`—mainly contain the `add annotation` and `move class` refactorings. We thus achieve our third validation step through dedicated micro-benchmarking. We first build a micro-benchmark to check the effect that every encountered annotation may have on energy consumption. We thus considered each of (`@override`, `@SuppressWarnings("unchecked")` and `@SuppressWarnings("unused")`). We then ran hundreds of millions of times each, on classes, methods and variables to check whether it has an effect on the energy consumption. The results—as expected—did not have any effect (about 1% difference that we cannot consider due to CPU energy variations seen in Chapter 4) on energy consumption, as annotations are not supposed to have a substantial impact on the generated bytecode that would be executed by the JVM. This would invalidate the fact that the observed energy consumption difference is mainly related to the `add annotation` refactoring in the commits that only contain this type of refactoring, such as `#827717`, `#b9dfbc`, and #298b7a. The second micro-benchmark concerns the move class refactoring, where we measured the energy consumption for several scenarios, after moving some classes/interfaces around and reorganizing the structure of the micro-benchmark. The results showed a difference in energy consumption of up to 8%, with an average standard deviation of 5%. The move class refactoring—which is often accompanied with the rename refactorings—indicates a code reorganization that might have an impact. While the observed impact through the JMH experiments or with micro-benchmarking might not be substantial, it would be beneficial to be aware that restructuring/reorganizing a project could have an impact on energy consumption, and thus compare the before/after energy consumptions to track that effect. Unfortunately, we could not detect any specific pattern or guidelines on when the code reorganization or restructuring would impact positively or negatively the energy consumption. Hence, we can only faithfully retain the commit #45bf2d of the Gson project among the commits of Table 6.3, where the 30 move class refactoring could have been responsible of 7% of energy consumption difference with a standard deviation of 5%.

We finally conclude that structure-oriented refactoring has no substantial impact on the energy consumption of the main functionality of 7 projects that have been existing for at least 5 years with a total of 16,046 commits. We argue that it could be applied to improve the code quality with no negative impact on software energy consumption. Although, comparing the energy consumption before and after the changes is always a good practice to keep track of its evolution.

To answer RQ2, we conclude that code refactoring rules are mostly "safe" operations that have **no substantial impact** on software energy consumption. Developers should not fear structure-oriented refactorings, especially regarding how little is the impact they could have compared to the real energy consumption evolution of projects, registered while answering RQ1.

6.4 Threats to Validity

There are a couple of issues that can impact the accuracy of our results. First, our analysis highly depends on the RefactoringMiner tool and its ability to extract every single occurrence of each of the 55 refactorings it supports. Moreover, there are some other refactorings, not listed among the 55, that have not been extracted and thus considered in our study, especially those related to implementation and computation details and those that cannot be discovered automatically.

We also focused on running benchmarks that last for many seconds (around 150 sec for Gson, 450 sec for XStream, 330 sec for OkHttp, 290 sec for Google-Http, 780 sec for JGraphT, 720 sec for JFlex, and 600 sec for Eclipse Collections), so we can obtain trustful and robust evaluations of the potential impact of changes between commits with an overall continuous execution time of experiments that exceeded 100 hours. Yet, as Intel RAPL only measures CPU and DRAM energy consumption, we built our experiments to be CPU- and RAM-intensive and tried to reduce the I/O and network access as those cannot be properly measured. For example, we used a local minimal HTTP server for OkHttp and Google-Http experiments to reduce the network impact.

To reduce the statistical uncertainty, we use the bootstrapping method to compute the mean of many generated subsets to simulate thousands of random sets of experiments from the total set of registered values.

The manual steps in our study remain the design of the JMH benchmarks and some checks of the git diffs. In the first case, we tried to write benchmarks that stress the main purpose or functionality of each project, so we can ensure that the comparison is based on the same functionalities that are available on all commits and versions. While this is moderately easy for some projects, such as Gson or XStream, it is much more complicated for other projects, such as Eclipse Collections where many collections and operations are available and can change. We tried in this case to cover many functionalities that are available in most commits, even if it requires some adjustments and adaptation when projects are restructured / reorganized between versions. Regarding git diff, we gave the major importance to the commits with the most impact, as it is not possible to meticulously check all the changes on all the selected commits. Another threat may be related to our selection of the commits with the most refactoring to have a reasonable execution time. Even if selected commits are most likely to be the most impactful.

How generalizable are our results? Based on the results of 7 open-source projects that have existed for at least 5 years, we believe that our results about the limited impact that have structure-oriented code refactorings on software energy consumption can be generalized, due to the high number or covered commits and refactorings, at least for the 55 refactorings extracted by RefactoringMiner. We also noticed that some projects tend to reduce their energy consumption, but this observation cannot necessarily be generalized to all projects.

6.5 Summary

This chapter describes an investigation of the effective impact of code refactoring on software energy consumption. We analysed 7 open-source Java projects and extracted 55 possible types of refactorings over all the commits, with more than 10k commits. We then selected the commits with the most refactorings and evaluated the impact that those refactorings could have on the energy consumption. This process ensures the evaluation of the effective impact that refactoring has for established projects that have existed for at least 5 years.

Overall, our results showed that structure-oriented refactorings have no substantial impact on the energy consumption on Java server-side software. This means that structure-oriented code refactorings can be safely applied to improve the maintainability and readability of source code with no significant penalty on the energy consumption of Java projects. When it comes to reducing software energy consumption, we believe that developers' efforts should be directed towards other software aspects and implementation optimizations rather than structure-oriented refactorings. For the Gson project, we noticed that even the commits with a lot of refactorings have no effective impact on the evolution of software energy consumption. However, the energy consumption of the Json serialization/deserialization features decreased by 4-fold in 3 years and 5-fold in 12 years. This highlights that the reduction in energy consumption of the project over time, is not driven by refactorings.

We believe that our approach can also be used to study/discover other refactoring rules, and extend our results to alternative projects, maybe for other languages than Java. Most importantly, this should motivate future works to validate that refactorings can be safely applied with no side effect on energy consumption, yet investigate the commits and the nature of code changes that increase/decrease energy consumption.

Chapter 7

Reducing the Energy Consumption of Java Software I/O

The previous chapter showed that structure-oriented refactoring does not have a substantial impact on server Java applications. However, other developers' actions at source code level might have an impact on software energy consumption. For instance, the Java language is rich of native and third-party I/O APIs that most Java applications and software use. Such operations can even be considered core to most software as they allow the interaction with the user and its data in a non-volatile way. In this context, the impact of these I/O operations on energy consumption did not get as much attention. Of course, I/O operations are responsible for energy consumption at the level of the storage medium (HDD or SSD) but can also induce non-negligible costs on both performance and energy at the CPU level.

Hence, this chapter elaborates a detailed study with two main objectives. First we aim at assessing the energy consumption of several well-known I/O libraries methods, and investigate if different read/write methods can exhibit different energy consumption. The second objective is to validate the results of the first experiments on real Java projects by refactoring their default I/O methods and measuring the before/after energy consumption. The results showed that *i)* different I/O methods consume very different amounts of energy, such as NIO Channels that are 20% more efficient than other methods for read purposes *ii)* substituting the I/O method in a software by a more efficient one can save an important amount of energy.

The remainder of this chapter is organized as follows. Section 7.2 introduces the methodology (hardware, projects and experiments design) we adopted in this study. Section 7.3 analyzes several experiments we conducted to evaluate the energy consumption of Java I/O methods, as well as the results we observed during these experiments.

7.1 Overview

Energy efficiency of software systems is undoubtedly a major challenge for the software engineering community. Beyond state-of-the-art key performance indicators, such as latency, throughput, scalability or availability, energy consumption challenges developers to reach the best performances while minimizing the subsequent resource requirements. In this context, previous studies in the field have been focusing on the impact of algorithms [62, 91] and data structures [35, 52] to reduce the energy consumption in presence of computation-intensive applications. Nevertheless, little effort has been invested in the study of the energy efficiency of Java I/O libraries [127] for the purpose of data-intensive systems/applications. Data-intensive applications are expected to process huge amounts of data for different purposes, from big data analytics to online cloud microservices. While the hardware components tend to keep improving on storage capacity and throughput, it remains unclear if their software counterparts succeed to keep the pace and provide energy-efficient solutions to efficiently read and write data. This is particularly difficult in Java, which provides a vast ecosystem of built-in functions and third-parties libraries to interact with persistent storage. In addition to this rich ecosystem, I/O APIs may be subject to the emergence of other features, like asynchronous capabilities (NIO), which can exhibit a different energy footprint. We believe that guiding the developers in picking the most energy efficient I/O method is a key challenge to deliver sustainable software with no compromise on the performances.

In this area, previous works have already shown the substantial contribution of I/O to the global energy consumption of software. For example, Lyu et al. [85] indicated that 10% of the mobile battery drains due to I/O operations. The energy consumption of Java I/O APIs has also been compared by Rocha et al. [127]. They reported that the energy consumption can widely vary among these APIs, and that the most popular APIs are not always the most energy efficient. This study delivers interesting insights about some APIs, such as `java.io.In(Out)putStream` and `java.io.Reader(Writer)`, as well as their inherited classes. However, it lacks some considerations for major I/O APIs, such as channels (`Java.nio.FileChannel`) or other popular third-party libraries (*e.g.*, Apache, Google Guava), it also did not consider different usage profiles of I/O, and failed to reproduce the experiments on a realistic Java project.

In this chapter, we therefore assess the energy consumption of 27 different I/O methods issued from multiple native and third-party libraries using micro-benchmarks and different workloads. These methods are tested and compared for different scenarios and use cases (read the whole file, read a file part-by-part, seek data from a file, write data in a file, using different buffer sizes, etc.). Concretely, the purpose of the study is to answer the following research questions:

RQ 1: How do I/O methods affect the energy consumption of a Java code?

RQ 2: Can we reduce the energy consumption of software by refactoring its I/O methods?

Our empirical exploration highlights the most energy-efficient methods for each use case. For instance, using channels for massive reads is usually 10–20% more energy efficient than other I/O methods. Moreover, we refactored the default I/O methods of well-known Java benchmarks and projects to effectively reduce their energy consumption.

Beyond answering these two research questions, the contributions of this chapter can be summarized as:

1. Elucidate the energetic behavior of 27 different I/O methods issued from multiple libraries, using several file sizes,
2. Identify the most energy-efficient methods for several read and write use-cases,
3. Model the energy consumption in regards to the buffer size for the buffered I/O methods,
4. Deliver insights and guidelines to summarize the results and conclusion of our experiments,
5. Investigate the potential gain of refactoring the default I/O methods of benchmarks and real java projects.

7.2 Methodology

To investigate the energy impact of different Java I/O libraries and methods, we conducted a wide set of experiments using several micro-benchmarks.

7.2.1 Environment Settings

In this chapter, we used the same test machine configuration used in Chapter 6. The machine is equipped with an Intel SSD Pro 5450s Series, with a capacity of 256 GB (up to 550 MB/s for reading and 500 MB/s for writing operations). We focused on measuring the CPU energy consumption using RAPL to evaluate the differences in energy consumption between the tested methods.

7.2.2 Experiments Design

Our experiments are structured in two steps. The first step has an exploratory nature: by testing and comparing numerous Java I/O libraries and methods to read and write data from files. To do so, we prepared text and binary files of several sizes. Table 7.1 summarizes the size on disk of each file type and the lines count $\times$ lines length for the equivalent text file of the same size. Next, we create micro-benchmarks for each I/O method, then run them for at least 30 times and measure the execution time and the CPU energy consumption. All read

Method	File Size	Lines count	Line length
Tiny	100 KB	1,000	100
Small	15 MB	100,000	150
Medium	200 MB	1,000,000	200
Medium-large	3.2 GB	8,000,000	400
Large	16 GB	40,000,000	400

Table 7.1: The list of file sizes.

benchmarks run the same function `consume` to compute a hash from the raw input data, to prevent the JIT from discarding some code and altering the expected behavior.

The second step aims at validating the results of the first step using some Java benchmarks from the *Computer Language Benchmarks Game* (CLBG): Fasta and K-nucleotide, but also using a real Java project (Zip4J) and a real Java API (Javax.Crypto). This validation is achieved by refactoring the read/write methods used in these benchmarks/projects by the ones that exhibited the lowest energy consumption in the previous step, and check whether the energy consumption of these source codes can be reduced.

Java I/O Libraries

We evaluated a wide set of read/write methods from several Java I/O libraries in our study. The full list of methods is provided in Table 7.2. The purpose is not to explore every sub-classes or sub-methods of the ones present in Table 7.2. For example, we do not test every method that extends `InputStream`, such as `PushbackInputStream` or `ByteArrayInputStream`. Such methods have already been compared in Rocha et al.'s study [127]. The purpose is rather to study and compare different methods issued from different classes, libraries, and even famous third-party solutions. Table 7.2 reports on 3 different purposes: "Read" to read data, "Seek" for accessing some data at a specific position and "Write" to write data. "ReadAll" is a particular case of Read where we read the content of the whole file at once. Most of the used classes are issued from `java.io` and `java.nio` (NIO). NIO is for non-blocking I/O—*i.e.*, by handling the I/O operations in a non-blocking way using buffers, channels, etc.

Some of the methods in Table 7.2 can use a buffer, which is a memory block of a given size (usually 8,192 bytes). At the opposite of an array or a list, a buffer has a limit that differs from its capacity. This enforces the ability to have a variable size up to the capacity, which is the maximum it can handle. Moreover, the buffer offers a built-in way to read or write the next element, easing sequential processing, and a way to save the current position for later reset (mark and position functions, respectively). We note that the versions of the used APACHE and GUAVA libraries are 2.8.0 and 23.0 respectively.

Class	Acronym	Method	Purpose	Availability	Description
java.io.InputStreamReader	IOSTREAM	read(Byte[])	Read	JDK 1.0	InputStreamReader is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset
java.io.OutputStreamWriter	IOSTREAM	write(String)	Write	JDK 1.0	OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset

java.io.FileInputStream	IOSTREAM	Skip(long)
		readAllBytes()
			Seek
			ReadAll
				JDK 1.0	A FileInputStream obtains input bytes from a file in a file system. It is meant for reading streams of raw bytes such as image data

java.io.BufferedInputStream	BIOSTREAM	readLine()	Read	JDK 1.0	A BufferedInputStream adds the ability to buffer the input and to support the limit, in addition to the mark and reset methods
java.io.BufferedOutpuStream	BIOSTREAM	write(String)	Write	JDK 1.0	It Implements a buffered output stream. It can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written
java.io.FileReader	FILEREADER	read(char[])	Read	JDK 1.1	FileReader is meant for reading streams of characters
java.io.FileWriter	FILEWRITER	write(String,int,int)	Write	JDK 1.1	FileWriter is meant for writing streams of characters
java.io.BufferedReader	BFILEREADER	readLine()	Read	JDK 1.1	Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines
java.io.BufferedWriter	BFILEWRITER	write(String)	Write	JDK 1.1	Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings

java.nio.channels.FileChannel	CHANNEL	read(ByteBuffer)
		write(ByteBuffer)
		position(long)
			Read
			Write
			Seek
				JDK 1.4	A Filechannel is a SeekableByteChannel that is connected to a file. It has a current position within its file which can be both queried and modified

java.nio.FileChannel	OMCHANNEL	map(mode,long,long)	Read	JDK 1.4	Maps a region of this channel's file directly into memory

java.nio.Files	NIOF	readLine()
		write(String)
		readAllLines(Path)
			Read
			Write
			ReadAll
				JDK 1.7	This class consists exclusively of static methods that operate on files, directories, or other types of files. It delegates to the associated file system provider to perform the file operations

java.util.Scanner	SCANNER	nextLine()	Read	JDK 1.5	A simple text scanner which can parse primitive types and strings using regular expressions

java.io.RandomAccesFile	RAF	readLine()
		writeBytes(String)
		seek(long)
			Read
			Write
			Seek
				JDK 1.0	A random access file behaves like a large array of bytes stored in the file system with a pointer into the implied array, used for both read and write operations

apache.commons.io.FileUtils	APACHE	readFileToStr(File,Ch)
		write(File,List,Bool)
		readFileByteArray(F, Ch)
			Read
			Write
			ReadAll
				NA	Apache General file manipulation utilities. It offers reading, writing, and much more operations

google.common.io.CharSource
google.common.io.CharSink
google.common.io.Files
	GUAVA	readLine()
		write(String)
		readLines(file, Charset)
			Read
			Write
			ReadAll
				NA	Google library that provides utility methods for working with files, including reading and writing operations.

Table 7.2: The list of the studied I/O classes and methods

Real Benchmarks

The impact that I/O operations could have on the global consumption mainly depends on the type of software. This impact could be high for some applications, such as file compression or serialization applications, while it can be much lower for other classes of applications that perform much less I/O operations. Hence, we choose benchmarks that stress the usage of I/O operations to confirm the results of the micro-benchmarks and clearly spot the differences in energy consumption that such operations can cause for real applications.

First, we choose 2 I/O focused benchmarks from the *Computer Language Benchmarks Game*: Fasta, and K-nucleotide. The first one generates DNA sequences (CGAT), by weighted random selection from 2 alphabets and outputs the results in a file. The second benchmark reads line-by-line the Fasta format of the previous file, extracts the DNA sequence number Three, and updates a hash-table of k-nucleotide keys to count values within a particular reading-frame.

We refactor the write and read methods from the Fasta and K-nucleotide with the methods that exhibited the best energy efficiency from our micro-benchmarks. The purpose is to check if we can reduce the energetic impact of those benchmarks with only the refactoring of the I/O method.

Moreover, we apply the same refactoring process on the read method of real Java projects: Zip4J[1] and Javax.Crypto.[2] The first project is a Java library that offers many operations for handling zips and streams. As far as we know, it is the only Java library with support for zip encryption, apart from several other features. The second project is a Java API that delivers many classes and interfaces for cryptographic operations. Refactoring the I/O operations in this context aims to deliver a more realistic feedback on the energy impact of I/O on a library that uses a fair amount of I/O operations, such as Zip4J and Crypto.

7.3 Experiments and Results

In this section, we expose the results of our experiments in order to answer our research questions.

7.3.1 Behavior of I/O Methods

In this part, we study the behavior of the considered I/O methods used in the micro-benchmarks. One benchmark has been written for each method of Table 7.2. Then, experiments are run with each benchmark and for several file sizes (cf Table 7.1).

[1]https://github.com/srikanth-lingala/zip4j
[2]https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html

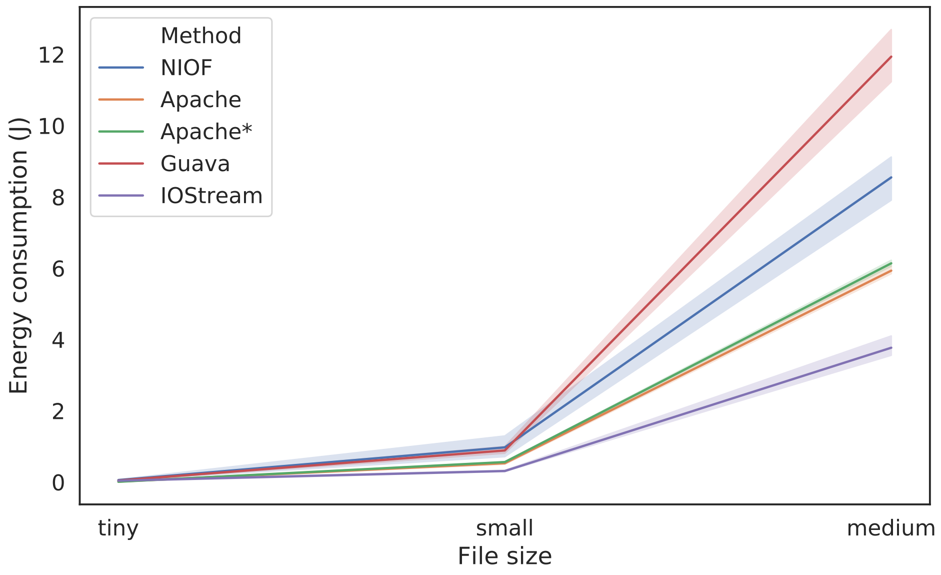


Figure 7.1: Energy consumption to read the whole content of files.

Reading the Whole File at Once

This method is limited by the memory size—*i.e.*, it can only be applied on small files where the RAM size would allow loading the content of the whole file at once. Some of the classes mentioned in Table 7.2 natively offer the possibility to read the whole content of a file (ReadAll methods from IOSTREAM, APACHE, NIOF, GUAVA). While we believe that this way to read files is not compatible with all needs and thus is not the most interesting, we still run a quick comparison of the available methods using the medium file at best due to the memory limitation. Figure 7.1 depicts the energy consumption of 5 native methods to read the whole content of binary and text (strings or lines) files issued from 4 classes. It shows that the `InputStream.readAllBytes()` method (IOSTREAM) is the most energy efficient, followed by APACHE `ReadFileToByteArray(File, Charset)` or `ReadFileToString(File)`. NIO `Files.readAllLines(Path)` and GUAVA `readLines(File, Charset)` methods consumed the most energy among the 5 tested methods.

Despite reading being limited by the file sizes, we can still notice a substantial gain in energy consumption with the appropriate method and use case. In fact, IOSTREAM consumed 4 times less energy compared to GUAVA and 3 times less compared to NIOF, for a medium binary file. It was the most efficient way to read whole files among the 5 methods and is 60% more efficient than APACHE to read binary files for medium file sizes. To natively read text files, APACHE* was the most efficient way to do it, 40% more efficient than NIOF and 100% more efficient than GUAVA.

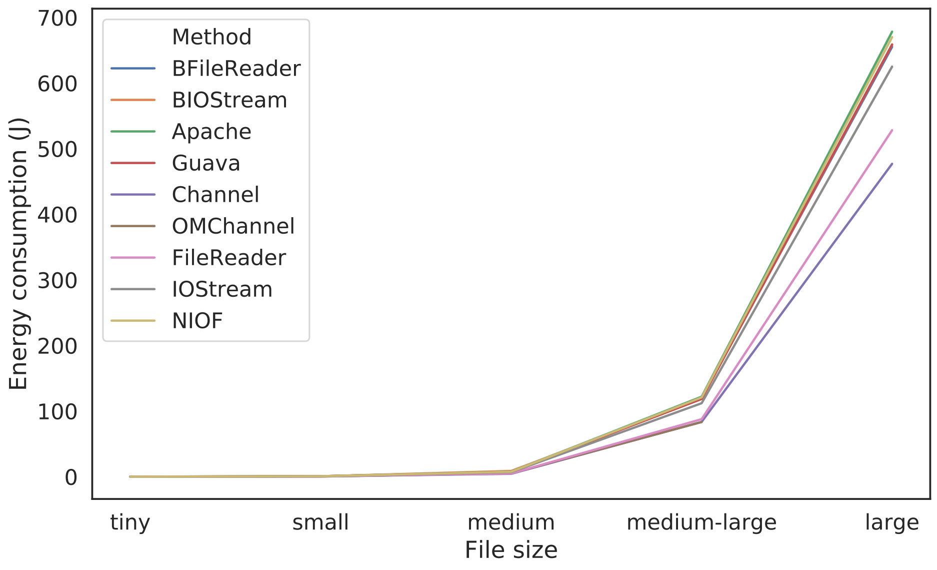


Figure 7.2: Energy consumption to read files by chunks for several file sizes.

Reading the Whole File by Chunks

This way of reading is much more flexible and bypasses the memory size limitation by reading the file by chunks. This is generally achieved by using a buffer or an array of a given size that slides through the file to read input data. Many libraries offer functions to natively achieve this. We thus run read methods from Table 7.2 with different file sizes and assess their energy consumption. Figure 7.2 overviews the energy consumption of all tested read methods on several file sizes. The observed difference in energy consumption seems to be very small for our tiny, small, and medium file sizes, but the lines become clearly distinguishable as the file sizes grow. We note that two other read methods (RAF and Scanner) have been tested, but are not depicted in the figure for readability reasons. Our experiments in Table 7.3 showed that different input methods consume different energy. The most extravagant case is RANDOMACCESSFILE where we noticed an extra energy consumption, even for tiny and small files. This extra energy becomes very important for bigger files (up to 200 times more energy). The SCANNER read is the second noticeable method that gave much worse results, compared to the others for all file sizes (more than 4 times more energy consumed with Scanner).

The other results of Table 7.3 are much closer, with methods that give very similar results for all file sizes (APACHE, BFILEREADER, BIOSTREAM, GUAVA, NIOF). The clear winners are CHANNEL and ONMEMORYCHANNEL that consumed the least energy among all methods and across all file sizes.[3] Along our experiments, NIO CHANNELS consumed up to 20% less energy, compared to the average and about 10% to the second-best method (FILEREADER),

[3]ONMEMORYCHANNEL could not be used on a large file due to memory limitation, as it uses memory mapping.

which constitute a substantial gain, especially for large files and applications that use a good amount of I/O.

Finally, we noticed that using the buffer for FILEREADER and INPUTSTREAM is not very beneficial. The buffered BFILEREADER and BIOSTREAM consumed up to 10% more energy for large files.

Method	**Tiny**		**Small**		**Medium**		**Medium-Large**		**Large**	
	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
Apache	0.03	1.9	0.68	58	7.7	724	122	$11.5 * 10^3$	678	$62 * 10^3$
BFileReader	0.01	1	0.69	61	7.7	714	119	$11.3 * 10^3$	655	$60 * 10^3$
BIOStream	0.01	1	0.7	60	8.6	748	118	$11.2 * 10^3$	658	$60 * 10^3$
Channel	**0.02**	1.5	**0.32**	27	**4.4**	434	**83**	$7.9 * 10^3$	**476**	$43 * 10^3$
FileReader	0.01	1	0.43	35	5	500	90	$9 * 10^3$	568	$50 * 10^3$
Guava	0.03	1.8	0.67	57	7.4	707	118	$11.2 * 10^3$	658	$60 * 10^3$
IOStream	0.03	1	0.49	44	7	683	112	$10.9 * 10^3$	625	$58 * 10^3$
NIOF	0.01	1.1	0.7	58	7.5	716	120	$11.5 * 10^3$	670	$62 * 10^3$
OMChannel	**0.01**	1	**0.3**	25	**5.1**	488	**83**	$7.9 * 10^3$	NA	NA
RAF	1.04	86	120	1200	1528	157062	24548	$25 * 10^5$	$1.2 * 10^5$	$13 * 10^6$
Scanner	0.1	8	3.1	230	19.4	1697	563	$5 * 10^4$	2893	$26 * 10^4$

Table 7.3: Energy consumption (joules) and execution time (ms) for reading files of different sizes by chunks.

Figure 7.3 exposes more visually the results of Table 7.3 for a large file. The violin plots show very stable energy consumption values with very small standard deviations. This reports on robust experiments, but also validates the results of Table 7.3, as all results are tightly centered around the average/median. The figure also allows to establish an easier comparison of the read methods issued from the different classes (RAF and SCANNER have been excluded for a better visualization, OMCHANNEL is not applicable for large files).

Figure 7.4 depicts the average of read data per joule for each method, while reading the large file. This confirms that the CHANNEL is more efficient and reads more data per joule. In fact, the CHANNEL read method reads about 35 MB/j. That is 5 MB/j more efficient than FILEREADER and 10 MB/j better than the average. RAF and SCANNER on the other hand are the least efficient with 5 MB/j and less than 1 MB/j, respectively.

One more question we wanted to address is: what is the most energy-efficient way to read data from a file if the memory size and the JVM heap size are sufficient to load the whole file? If we compare the necessary energy consumption to read our medium file in Figure 7.1 and Table 7.3, we notice that reading the whole file at once with IOSTREAM consumed about 3.8 joules. That is at least 13% less consumed energy than any other method in Table 7.3 (4.4 joules being the lowest). This indicates that reading the whole file can be more energy efficient than reading it by chunks, when possible.

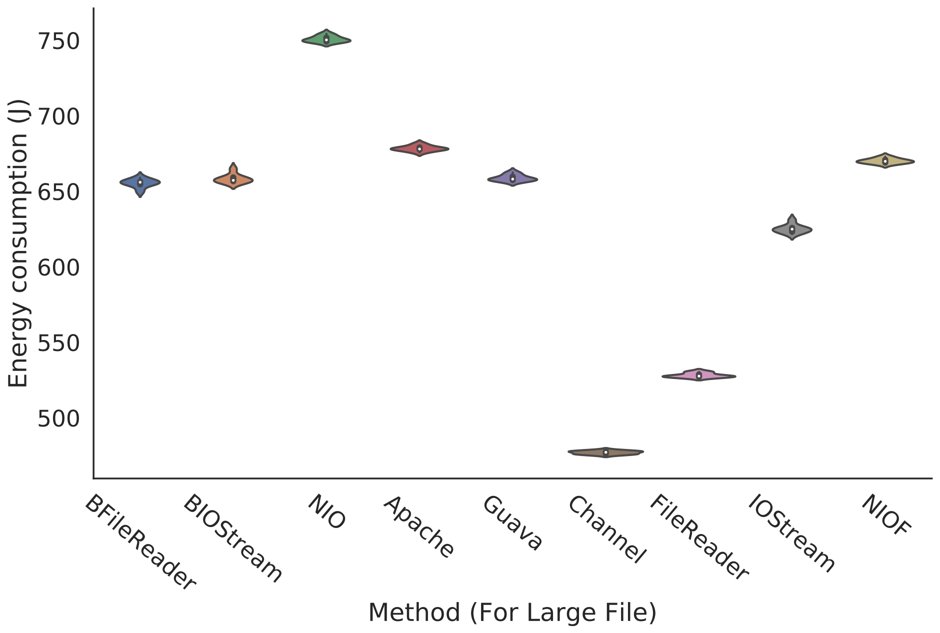


Figure 7.3: Energy consumption of read methods for a large file.

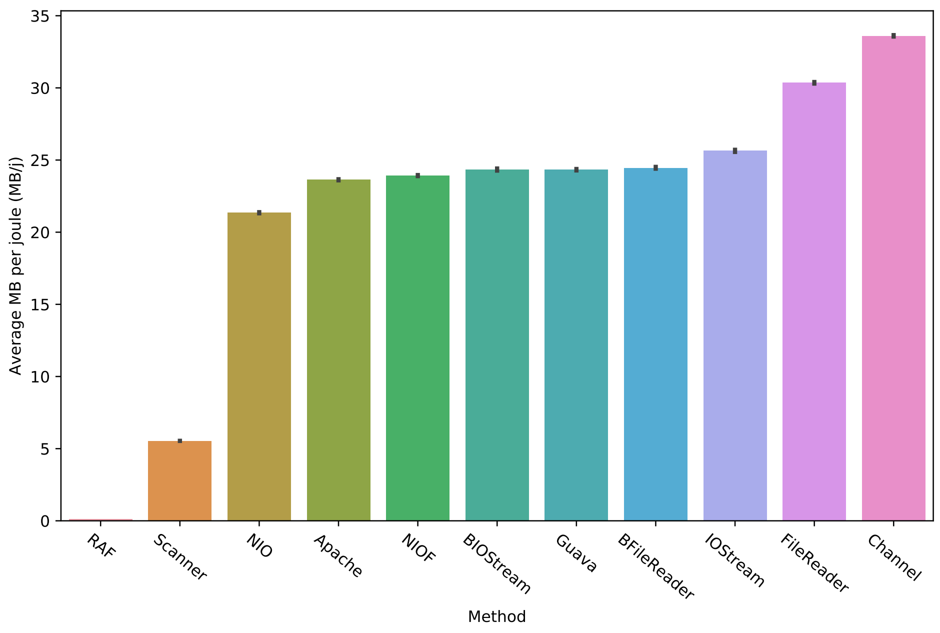


Figure 7.4: The average of read data per joule for each method.

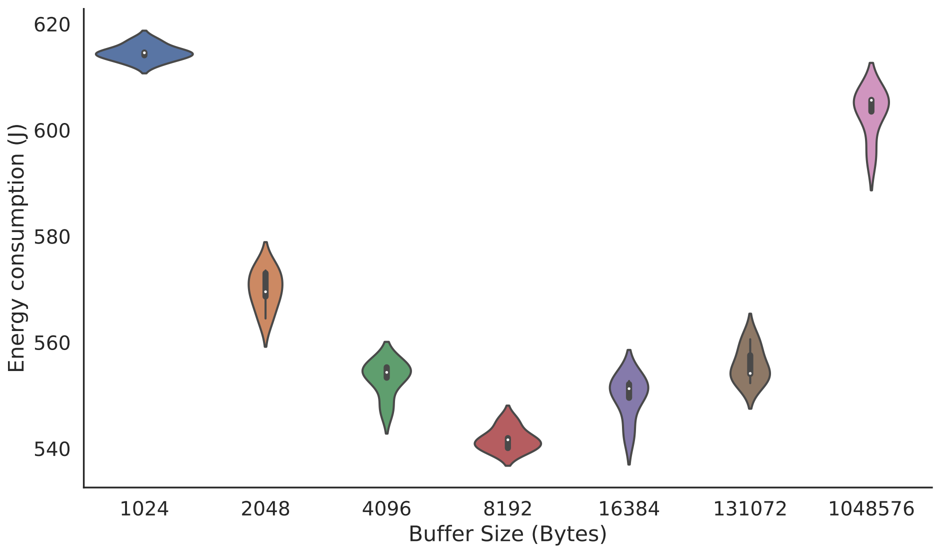


Figure 7.5: Energy consumption to read files with different buffer sizes.

Buffer Size Many of the previous methods require or use a buffer to perform the reading operation (BFileReader, Channel, OMChannel, RAF, etc.). This buffer size may vary and can be set by the developer. In this experiment, we check the effect of the buffer size on the energy consumption.

Thus, we run many executions of the BIOStream `BufferedInputStream` read operation, and we vary the buffer size from its default value used for the previous experiments (8,192 bytes).

This difference is not noticeable for small files, but can be clearly seen in Figure 7.5 for a large file. This figure illustrates the measured energy consumption for each buffer size. We notice a parabolic shape of the values of energy consumption, approximately centered around 8192 bytes. The difference in energy consumption is very small (less than 3%) going to the nearest values to 8,192 bytes (4,096 bytes and 16,384 bytes). However, this difference grows much higher for further small or big values of the buffer size. Here, up to 13–15% more energy consumption for the buffer sizes 1,048,576 bytes and 1,024 bytes, respectively. Hence, the buffer size should not be too small or too big for a better energy efficiency.

Seeking Specific Data From a File

This represents the third possibility to read data from a file. It consists of moving a cursor or a pointer within a binary file, and reading an amount of data. This way is very interesting to access some of the data within the file without reading the whole file, especially for very large files. One potential application relates to database files that contain an index, such as SQLite, where we need to access the data pointed by the index without reading the whole database.

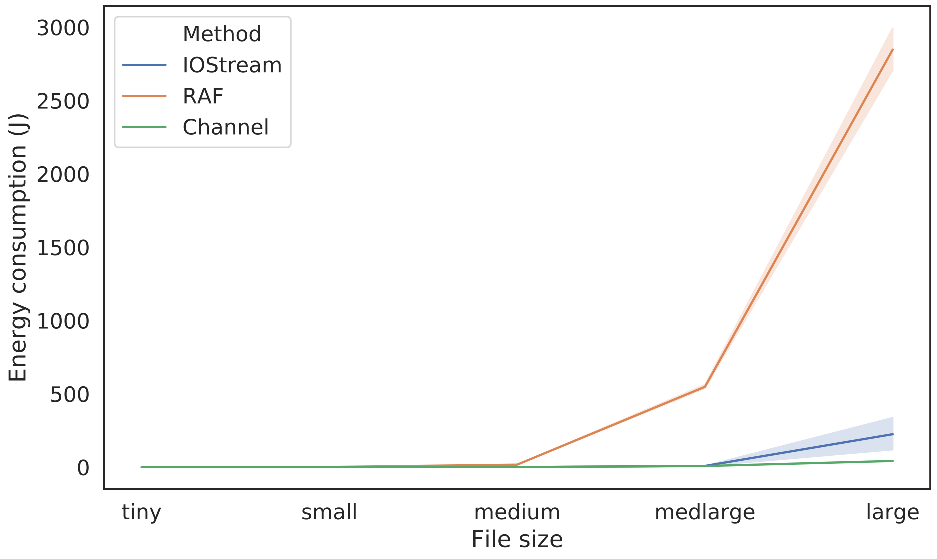


Figure 7.6: Energy consumption to seek data from files of different sizes.

Thus, we ran multiple experiments to compare the available seek methods for Table 7.2 and their energy consumption. To do so, we access and read different data at numerous positions within the files of multiple sizes. Each time, we read 100 bytes and move forward with 10,000 bytes to read the next 100 bytes, until the end of the file.

Table 7.4 and Figure 7.6 show the difference in energy consumption between the 3 different methods issued from CHANNELS, IOSTREAM, and RAF. Here again, the results show that using the NIO channels is the most energy-efficient way to seek data from files. This efficiency is mostly noticeable for a large file with a high number of seek operations, where the CHANNELS position method is 5 times faster than IOSTREAM skip method. On the other hand, the RAF seek method is the slowest and the most energy consuming. It consumed 67 times more energy than NIO Channels for this experiment on a large file.

Method	**Tiny**		**Small**		**Medium**		**Medium-Large**		**Large**	
	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
Channel	0.01	1.2	0.07	7.6	0.5	49.5	7.7	766	**42**	$4.1 * 10^3$
IOStream	0.01	1.1	0.06	7.6	0.6	62	8.3	814	225	$8.2 * 10^3$
RAF	0.01	1.6	1.2	111	17	1693	547	$5.3 * 10^4$	2847	$2.7 * 10^5$

Table 7.4: Energy consumption (joules) and execution time (ms) for seeking data from different files.

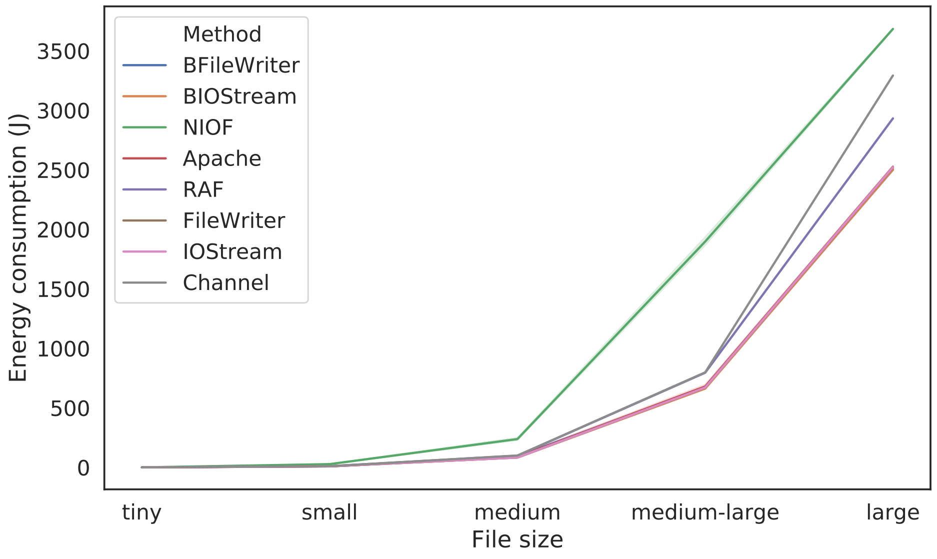


Figure 7.7: Energy consumption to write files for several file sizes.

Writing a File

The write operation consists of physically writing data on a file in its disk location. Like reading, there are plenty of classes that offer writing methods to achieve that purpose. Therefore, we assess the energy consumption of the write methods from Table 7.2 and try to identify the "good" and "bad" methods regarding the energy efficiency with different file sizes. Writing a file of a specific size is achieved by generating random text data of the appropriate size (LinesCount $\times$ LineLength) as described in Table 7.1 and writing it into a file.

Figure 7.7 depicts the evolution of the consumed energy by the different methods with growing file sizes (GUAVA has been omitted for a better visualization), while Table 7.5 gives the detailed energy consumption and the execution time of each method, and for each file size.

The results highlight that 4 of the tested methods are way more energy-consuming than the others. First, GUAVA's write method registered the worst energy efficiency, compared to all other methods and, for all file sizes, up to 6 times more energy consumption compared to the best-tested methods. Second, the NIOF writing operation consumed 50–100% more energy across all file sizes. Third, the NIO channels that gave the absolute best results for file reading are less efficient for the writing operation with 30% more energy consumption for large files. RAF is in the fourth position of the methods that performed worse than the others for write operations. It consumed 15% more energy compared to the best method.

The remaining methods (APACHE, BFILEWRITER, BIOSTREAM, FILEWRITER, IOSTREAM) gave very similar results, and were the most energy efficient methods to write data into files.

Figure 7.8 delivers a better comparison of the energy consumption of the write methods for a large file.

Method	**Tiny**		**Small**		**Medium**		**Medium-Large**		**Large**	
	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
Apache	0.20	13.3	8.4	760	8.7	$7.8 * 10^3$	679	$6.2 * 10^4$	2526	$2.3 * 10^5$
BFileWriter	0.24	15.1	9.5	794	8.6	$7.8 * 10^3$	$6.1 * 10^4$	$11.3 * 10^3$	2501	$2.3 * 10^5$
BIOStream	0.22	13.7	8.9	770	82	$7.5 * 10^3$	662	$6.0 * 10^4$	2502	$2.3 * 10^5$
Channel	0.23	14.5	10	919	99	$9.1 * 10^3$	798	$7.3 * 10^4$	3293	$2.8 * 10^5$
FileWriter	0.15	10.5	8.4	764	83	$7.6 * 10^3$	669	$6.1 * 10^4$	2518	$2.3 * 10^5$
Guava	1.18	80.7	95	6882	962	$7.1 * 10^4$	7507	$5.5 * 10^5$	15592	$1.7 * 10^6$
IOStream	0.12	9	8.5	765	83	$7.5 * 10^3$	671	$6.1 * 10^4$	2522	$2.3 * 10^5$
NIOF	0.32	22.3	27	1714	238	$1.5 * 10^4$	1897	$1.2 * 10^5$	3684	$3.4 * 10^5$
RAF	0.14	10.6	10	920	98	$9 * 10^3$	795	$7.3 * 10^4$	2932	$2.6 * 10^5$

Table 7.5: Energy consumption (joules) and execution time (ms) for writing files of different sizes by chunks.

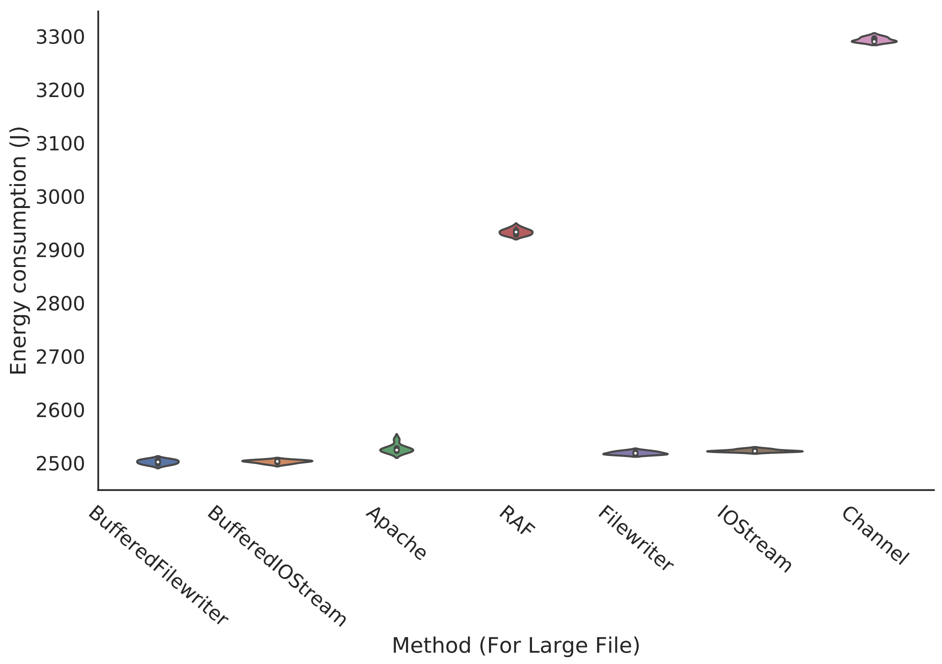


Figure 7.8: Energy consumption of write methods for a large file.

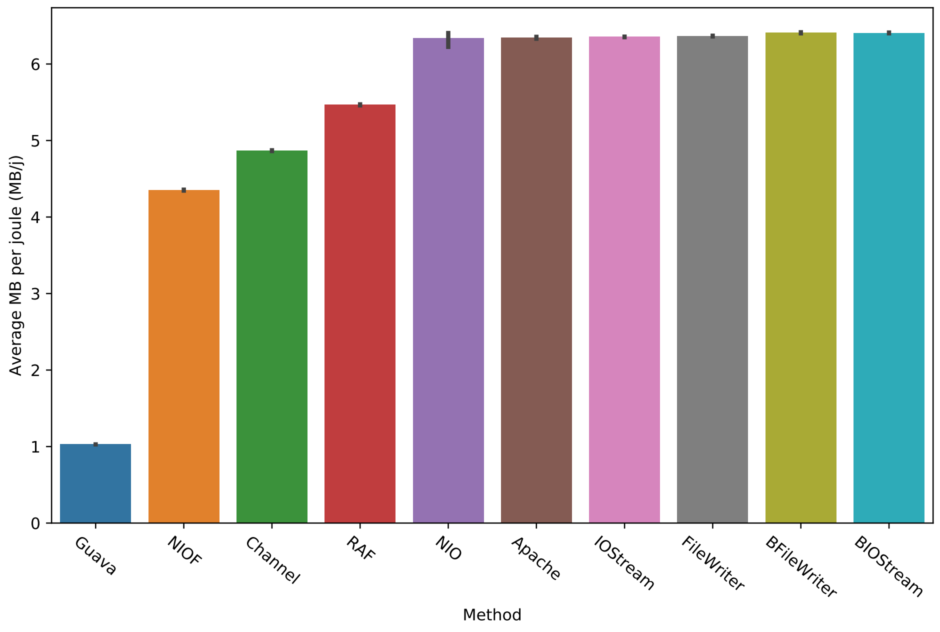


Figure 7.9: The average of written data per joule for each method.

One thing we noticed by comparing Figure 7.3 and Figure 7.8 is that writing a file consumes more energy than reading a file of the same size. In fact, we only needed less than 500 joules to read our large file with the most efficient read method, while the least-consuming write method needs 5 times that amount of energy (2500 joules) to write a file of the same size. This can be clearly seen through Figure 7.9 that shows the average written data per joule for each method. The maximum value we observe for the most energy efficient write operations is 7 MB/j. This is much lower than the values obtained for read operations (up to 35 MB/j in Figure 7.4).

To answer RQ1, we conclude that using different I/O methods can alter the software energy consumption. Our experiments delivered some insights and guidelines that can be summarized as:

1. For read methods, using the class `nio.FileChannel` proved to be the most appropriate choice to consume the least amount of energy while reading files of different sizes. It was at least 10–20% more energy efficient;

2. SCANNER and RAF reported on a very high energy consumption, compared to the other methods and should thus be carefully used, if not avoided, for data reading purposes;

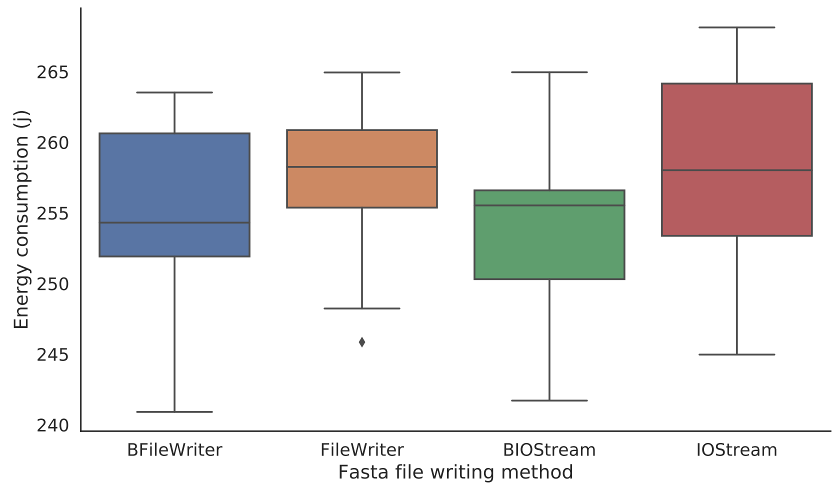
3. For methods that use buffers, using buffer sizes that are too big or too small compared to the default size of 8,192 bytes may introduce an extra cost in energy consumption;
4. Reading the whole file at once is limited by the file/memory size, but can be very energy efficient;
5. For write operations, many methods reported a similar energy consumption, but other methods, such as GUAVA or NIOF consumed more energy, and should thus be avoided.

7.3.2 Refactoring I/O Methods

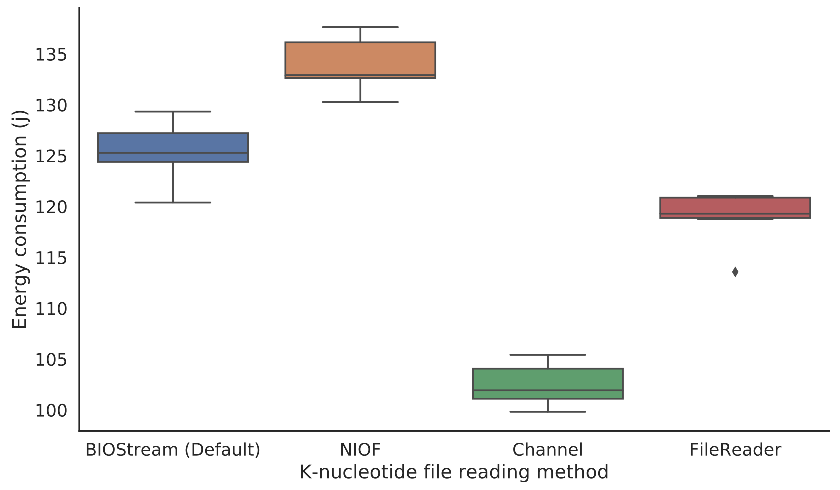
In this part, we aim at answering the second research question by refactoring and comparing multiple I/O methods on 4 Java projects presented in Figure 7.10: Fasta, K-nucleotide, Zip4J, and Javax.Crypto. Thus, we refactor the write method from the Fasta program with the methods that reported on a low energy consumption from the previous step to save a 1.5 GB of generated nucleotides. We then assess the energy consumption with each of the methods used, for 30 executions each.

Figure 7.10a illustrates the boxplots that summarize the energy consumption of each method's execution. As seen in the previous section, the write methods are almost similar and consume approximately the same amount of energy (around 255 joules). The written Fasta data is used as an input file for the K-nucleotide program. Figure 7.10b depicts the energy consumption of the nucleotide read operations using the methods that we used as substitutes to the default BIOSTREAM read method (the methods that performed best on the micro-benchmarks). Just like in our previous micro-benchmark experiments, the figure shows that using CHANNELS results in a lower energy consumption (15% less energy consumption using CHANNELS compared to the default BIOSTREAM).

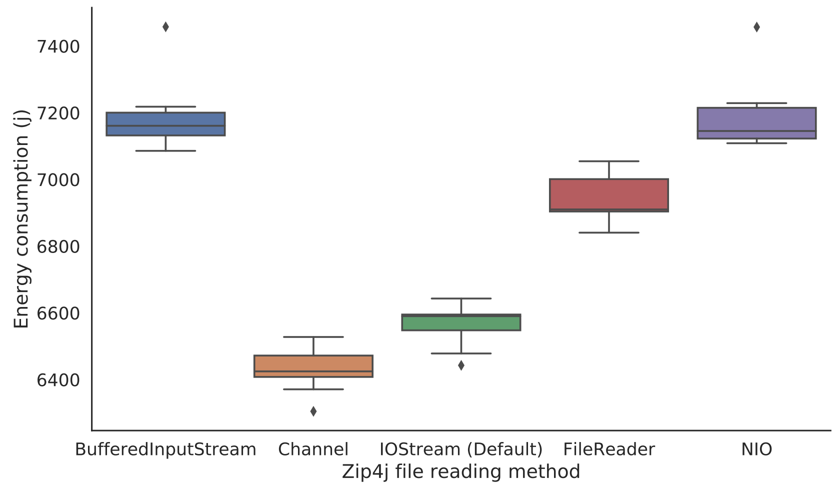
This result can be confirmed by Figure 7.10c in which we refactored the reading method of the Java zipping library with other read methods susceptible of having an equivalent or better energy efficiency. Then, we ran experiments where we zipped our large file and saved the result. Here again, using CHANNELS causes a reduction of 3% in energy consumption, compared to the default INPUTSTREAM method. The gain of using channels is only 3% in this experiment because the file reading phase only covers 10–15% of the total execution time of the zipping process. Finally, Figure 7.10d represents another confirmation of the energy efficiency of using CHANNELS for file reading purposes. In this example, using CHANNELS was at least 30% more energy efficient than FILEREADER and IOSTREAM (50% more energy efficient than BIOSTREAM and NIOF) to decrypt a 3 GB file using the Crypto API with the AES algorithm and a 16 bytes key.



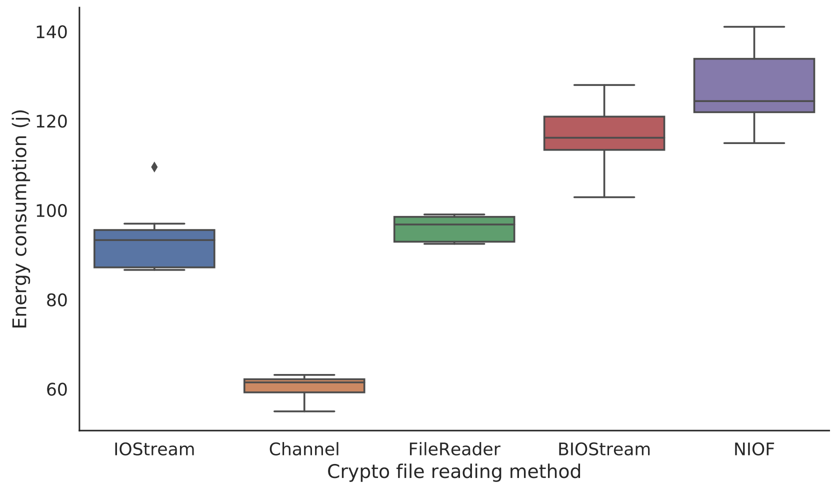
(a) Energy consumption of Fasta to generate and write 1.5 GB of nucleotide.



(b) Energy consumption of K-nucleotide to to read the data generated by Fasta.



(c) Energy consumption of Zip4J to zip our large file with different read methods.



(d) Energy consumption of the Crypto library to decrypt a 3 GB file with different read methods.

Figure 7.10: Comparing the energy consumption of Fasta with multiple write methods, K-nucleotide with multiple method to read the Fasta generated file, Zip4j to read and zip our large file using different read methods, and the Crypto library to decrypt a 3 GB file.

To answer RQ2, we proved that we can reduce the energy consumption of software and programs that run a substantial amount of I/O by choosing the right methods. Using NIO Channels proved to be very energy efficient here again on K-nucleotide, Zip4J, and Crypto, compared to other read methods.

7.4 Threats to Validity

The execution time and registered energy consumption for short tasks is one subtle threat to validity. In fact, some read and write experiments on tiny and small files are very fast, so we cannot assess the energy consumption faithfully. To overcome this issue, we constituted every execution of these fast experiments of many iterations. Most importantly, we focused all of our results analysis and conclusions on the experiments that last much longer, using our medium and large files. Java *Just-in-time* (JIT) compiler might constitute another threat to the validity of this work, especially for read operations in micro-benchmarks, if the read data is not used. Thus, we executed a hash method that consumes the read data similarly to a black-hole in JMH (*Java Microbenchmark Harness*). We also discarded the whole JVM instance between executions, so the JIT does not cache data between executions and alter the measures. We did not disable the JIT because the study would not reflect a real usage of I/O methods in realistic Java applications anymore.

The considered I/O libraries come with multiple read and write operations. To conduct this first study, we were obliged to select some of these methods. Our aim was to diversify our method selection to deal with both binary and text files, but we also wanted to select methods with similar signatures, so we can construct a fair and relevant comparison between the I/O libraries.

7.5 Summary

This chapter reports on an empirical investigation of the key differences in energy consumption of some famous Java I/O libraries and their read and write methods. Concretely, we assessed the energy consumption of 27 different methods using dedicated micro-benchmarks regarding several scenarios: read the whole file at once, read the file by chunks (with optimal buffer size), seek specific data within a file and write data to a file. Our experiments showed that not all read and write methods exhibit the same energy consumption while reading or writing data. On one hand, some methods can be very efficient, such as using NIO channels for read operations. On the other hand, other methods can be very inefficient, such as using *Random Access File* to read or write data.

To validate our results, we refactored/compared I/O methods on 4 real benchmarks and real Java projects with methods that registered a good energy efficiency with micro-benchmarks. We were able to reduce the energy consumption on three of them by using NIO Channels, achieving 15%, 3%, and 30% energy savings for K-nucleotide, Zip4j and Javax.Crypto, respectively.

Chapter 8

Conclusion

In this manuscript, we presented multiple contributions to understand and enhance software energy efficiency. First, we investigated developers' sensitivity, awareness and knowledge about SEC. The insights of this qualitative study helped us understand some major hurdles against GSD considerations. Among these hurdles, the lack of user-friendly, accurate and steady measurement tools. We also discussed in this document SEC variations problem and how to tune using some actionable parameters to conduct more accurate experiments. Once we had reviewed energy measurements, we stepped into some developers actions/decisions that can alter SEC. We focused on the JVM platforms and the Java language, and investigated the impact of multiple aspects (JVM configurations, Java code refactoring, Java I/O APIs) on the software energy consumption. The remainder of this chapter summarizes each of these contributions in Section 8.1. Section 8.2 and Section 8.3 introduce the main perspectives and future works stemming from the contributions of this thesis.

8.1 Contributions

The contributions of this thesis are summarized as follows.

On reducing the energy consumption of software: from hurdles to requirements. In this contribution, we conduct a qualitative study on 10 experienced software developers. The purpose of the study is to understand how developers feel about SEC, and what prevents a concrete introduction of GSD in software development within companies. Moreover, the work investigates developers tooling needs and requirements regarding SEC, and means to promote GSD within companies and among developers. Concretely, it investigates the following questions:

RQ 1: What are the hurdles that prevent the broader adoption of green software design?

RQ 2: What are developers' requirements in terms of tooling in an industrial context?

The contribution thus highlights many hurdles that prevent a wider adoption of green software design, including the lack of knowledge, awareness, time, tools and communication. It also discusses developers requirements for tools, such as the simplicity of interaction and integration with the current procedures (CI/CD), global scores and KPIs, etc. Finally, the study summarizes a set of implications for developers, decision makers, tool creators and researchers.

> Z.Ournani, R.Rouvoy, P.Rust, and J.Penhoat. On reducing the energy consumption of software: from hurdles to requirements. In Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '20, Bari, Italy. Association for Computing Machinery, 2020. ISBN: 9781450375801. DOI:10.1145/3382494.3410678. URL:https://doi.org/10.1145/3382494.3410678.

Another contribution derived from this work has been added to a book chapter, with examples on opportunities for developers to participate in SEC reduction and GSD evolution.

> J.Penhoat, M.S.Vaija, D.Phan-Huy, G.Gérard, Z.Ournani, D.Nussbaum, G.Dretsch, Q.Fousson, and M.Vautier. Green for ICT, Green by ICT, Green by Design. In Design Innovation and Network Architecture for the Future Internet. 96-121. Hershey, PA: IGI Global, 2021. http://doi:10.4018/978-1-7998-7646-5.ch004

Taming energy consumption variations in systems benchmarking. In this study, we investigate the phenomenon of variation when measuring the energy consumption of experiments. We discuss in this work multiple hardware and software factors that can amplify the variations of the recorded energy measures, with a focus on the following research questions:

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

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

RQ 3: Does the choice of the processor matter to mitigate the energy variation? and finally

RQ 4: What is the impact of the operating system on the energy variation?

This contribution highlights the significant impact that processor features could have on the energy consumption variation, compared to the benchmarking protocol or the operation system.

Finally, this study delivers multiple guidelines on controllable parameters that practitioners could easily tune to reduce the variations and conduct more steady/reproducible experiments.

Z.Ournani, M.C.Belgaid, R.Rouvoy, P.Rust, J.Penhoat, and L.Seinturier. Taming energy consumption variations in systems benchmarking. In Proceedings of the ACM/SPEC International Conference on Performance Engineering, ICPE '20, 36–47, Edmonton AB, Canada. Association for Computing Machinery, 2020. ISBN: 9781450369916. DOI:10.1145/3358960.3379142. URL:https://doi.org/10.1145/3358960. 3379142.

Evaluating the impact of Java virtual machines on energy consumption. In this contribution, we provide a deep evaluation of the impact of JVMs on the SEC. We expose through this study multiple experiments on hundreds of JVMs versions issued from several providers in order to answer 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?

The results show that depending on a software and a use-case, choosing the right JVM platform can drastically reduce the energy consumption. Moreover, setting a proper configuration of JIT and GC parameters can also significantly reduce the SEC.

Z.Ournani, M.C.Belgaid, R.Rouvoy, P.Rust, and J.Penhoat. Evaluating the impact of java virtual machines on energy consumption. En. In Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2021.

To complete this study, we deliver an open-source tool J-Referral.[1] It takes a software and a test script as inputs, run it over hundreds of JVM platforms/configurations and assess the energy consumption. At the end, it provides a full report with the least energy consuming options and how the software behaves for other configurations.

Tales from the code #1: the effective impact of code refactorings on software energy consumption. The research questions behind this contribution are:

RQ 1: How does the energy consumption of software evolve over time?

RQ 2: How do code refactorings contribute to the evolution of software energy consumption?

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

The purpose is thus to track the evolution of SEC of multiple projects that has existed for many years, and measure the impact of structure oriented code refactorings on this evolution. First, the study shows that the energy consumption of functionalities changes over time, with a trend to decrease for most of the server-side/desktop Java projects we tested. Moreover, the contribution show that structure oriented code refactorings have no substantial impact on the energy consumption and do not contribute much to the evolution of projects SEC.

Z.Ournani, R.Rouvoy, P.Rust, and J.Penhoat. Tales from the code #1: the effective impact of code refactorings on software energy consumption. In Proceedings of the 16th International Conference on Software Technologies, pages 34–46, 2021. ISBN :978-989-758-523-4. 3379142.

Evaluating the energy consumption of Java I/O APIs In this study, we assess the energy consumption of numerous Java I/O APIs. Concretely, we experiment multiple scenarios to Read, Write and Seek data and investigate the least energy consuming method among 27 methods issued from Java.IO, Java.NIO and third party APIs to answer the following questions:

RQ 1: How do I/O methods affect the energy consumption of a Java code?

RQ 2: Can we reduce the energy consumption of software by refactoring its I/O methods?

The results of our study show that some I/O methods are more adapted for some use cases and can drastically reduce SEC, such as NIO Channels for large files read purposes. Furthermore, refactoring the I/O methods on real projects based on the results of the study, proved to be beneficial and reduced the energy consumption by up to 30%.

Z.Ournani, R.Rouvoy, P.Rust, and J.Penhoat. Evaluating the energy consumption of java i/o apis. En. In Proceedings of the 37th International Conference on Software Maintenance and Evolution, 2021.

8.2 Short-Term Perspectives

Include developers in the GSD evolution Several studies have been conducted regarding software energy consumption and its efficiency. These studies have been mostly conducted by researchers with little to no interaction with developers. Consequently, the available tools and guidelines are not well designed to match developers' needs. We thus argue that future works should have tight implications on developers and users who could provide feedback from a different angle on how to enhance the energy efficiency of software. This has the advantage of providing knowledge that will be very likely beneficial to developers, but also explore new questions that they will raise accordingly.

Mining energy efficient patterns In chapter 6 we reviewed the evolution of energy consumption of some Java projects over the years. The study showed that the energy consumption of features tends to decrease across the versions. One major future contribution is to identify the exact changes that caused this decrease in energy consumption. Ultimately, the goal is to automatically mine atomic or basic changes that have previously decreased the energy consumption on multiple Github projects. This set of mined changes would be used to deduce energy efficient patterns that will be suggested to other projects to reduce the SEC. Moreover, these patterns would create a base of knowledge for multiple languages and environments, that developers could use upstream to produce energy efficient software.

Green Commits One other desired feature is the ability to assess the energy footprint of every new version, release and commit. The objective is to build context independent plug-ins that assess the energy consumption of a commit, and track the SEC across multiple commits. Therefore, developers would be able to monitor the evolution of SEC for each commit, and spot the changes that were responsible for increasing/decreasing the energy consumption. This could also be a part of the continuous integration process, where warnings should be raised if the "energy tests" fail (the SEC increases exceeding a threshold).

Automatic suggestions of green Java I/O methods We showed in Chapter 7 how using different I/O methods can substantially alter the energy consumption of Java programs for several use cases. For future works, we do see a massive reproduction of our results on a large set of projects. The purpose is to refactor the default I/O methods on multiple Github Java projects, assess the energy consumption, and open pull requests on those projects to enhance their energy efficiency. This could be achieved using an automated I/O methods referral tool that detects energy-consuming I/O methods and recommends more energy-efficient alternatives. Such a tool will be an easy way to substantially reduce the energy consumption if I/O intensive software. It will also help developers acquire some green coding habits.

8.3 Long-Term Perspectives

A Complete GSD tooling set One of the main requirements of developers to introduce GSD in their daily work is to have the necessary tooling to track and enhance the evolution of their software energy consumption. While some tools already exist to measure the energy consumption and assist developers at some code aspects such as the JVM platform or Java collections selection, this is still not enough to provide a satisfactory developers' experience. In fact, the purpose is to build a complete solution to assist developers in their green software design task while matching their requirements and working habits. Thus, tools should be language and context independent. They should allow to compute global scores/KPIs so developers could track the evolution of their software, and be able to go further to investigate the reasons behind improvements or drawbacks in energy consumption. Concretely, such a tooling set should cover all aspects of energy measurements, tracking and advising, allowing developers intervene at multiple levels of a software life-cycle.

Cloud energy consumption Cloud services are undoubtedly very attractive due to the flexibility and ease of deployment and/or use of services. These services are however poorly documented when it comes to energy usage and CO2 footprint. Probably due to some cloud providers' willingness to keep such data shady and gloomy for now, but also due to the absence of enough knowledge and means to establish accurate energy consumption reports of services across multiple levels of virtualization and pooling. With the rapid emergence of GSD and energy consumption awareness, future cloud services should adapt their solutions and offers to monitor the consumed energy and the emitted CO2 out of the service usage. Furthermore, users should be able to estimate their needs in energy, and choose/adjust the offers to reduce the carbon footprint. This will constitute an additional criteria that will increase the competition between providers, diversify the offers, and push towards improving the actual virtualization/pooling abstraction layers to offer a better monitoring of energy. All in favor of having energy efficient digital services.

Software LCA We advise applying a life cycle analysis (LCA) as it is the case for some hardware components to accurately assess software energy consumption and CO2 footprint. Applying such a process is quite different from the current measurements of a running software energy consumption. In fact, it includes a wider set of criteria and phases, starting with software development, where software vendors/providers should include the energy and CO2 footprint of the whole development process (planning, design, development, tests, etc.). Then, the energy consumption and CO2 footprint of the deployment/installation phase on one hand and the usage phase on the other hand. Finally, similarly to hardware, we should be able to estimate software end of life costs, including data migration, software abandonment and decommissioning, etc. Defining and using such a process will help estimating the real cost of a software. This is an important step towards sobriety, as it allows an accurate assessment of the energy consumption and CO2 footprint, taking into account all the necessary parameters (energy source, all hardware components, interaction with the surrounding environment, etc.).

Bibliography

[1] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999. ISBN 0201485672.

[2] *Pearson's Correlation Coefficient*. Springer Netherlands, 2008.

[3] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 30 years of software refactoring research: A systematic literature review. *CoRR*, abs/2007.02194, 2020. URL https://arxiv.org/abs/2007.02194.

[4] Bilge Acun, Phil Miller, and Laxmikant V. Kale. Variation Among Processors Under Turbo Boost in HPC Systems. In *Proc. of the 2016 International Conference on Supercomputing - ICS '16*. ACM Press, 2016.

[5] Tedis Agolli, Lori L. Pollock, and James Clause. Investigating decreasing energy usage in mobile apps via indistinguishable color changes. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 30–34. IEEE, 2017. doi: 10.1109/MOBILESoft. 2017.17. URL https://doi.org/10.1109/MOBILESoft.2017.17.

[6] Pedro Alonso, Rosa M. Badia, Jesus Labarta, Maria Barreda, Manuel F. Dolz, Rafael Mayo, Enrique S. Quintana-Ortí, and Ruym'n Reyes. Tools for power-energy modelling and analysis of parallel scientific applications. In *2012 41st International Conference on Parallel Processing*, pages 420–429, 2012. doi: 10.1109/ICPP.2012.57.

[7] Danilo S Alves, Lucio M Duarte, Davi Silva, and Paulo H M Maia. Experiments on Model-Based Software Energy Consumption Analysis. page 8, 2020.

[8] Anders S. G. Andrae and Tomas Edler. On global electricity usage of communication technology: Trends to 2030. *Challenges*, 6(1):117–157, 2015. ISSN 2078-1547. doi: 10.3390/ challe6010117. URL https://www.mdpi.com/2078-1547/6/1/117.

[9] Anders SG Andrae. New perspectives on internet electricity use in 2030. *Engineering and Applied Science Letters*, 3(2):19–31, 2020.

[10] H. Anwar, D. Pfahl, and S. N. Srirama. Evaluating the impact of code smell refactoring on the energy consumption of android applications. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 82–86, 2019.

[11] Hina Anwar, Dietmar Pfahl, and Satish N. Srirama. Evaluating the Impact of Code Smell Refactoring on the Energy Consumption of Android Applications. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 82–86, Kallithea-Chalkidiki, Greece, August 2019. IEEE. ISBN 978-1-72813-421-5. doi: 10.1109/SEAA.2019.00021.

[12] L. Ardito, G. Procaccianti, M. Torchiano, and A. Vetrò. Understanding green software development: A conceptual framework. *IT Professional*, 17(1):44–50, 2015.

[13] Maria Avgerinou, Paolo Bertoldi, and Luca Castellazzi. Trends in Data Centre Energy Consumption under the European Code of Conduct for Data Centre Energy Efficiency. *Energies*, 10(10):1470, September 2017. doi: 10.3390/en10101470.

[14] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks&mdash;summary and preliminary results. In *Proc. of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91. ACM, 1991.

[15] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-771-4. doi: 10.1145/1644893.1644927.

[16] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013. ISBN 978-3-319-04518-4. doi: 10.1007/978-3-319-04519-1_1.

[17] A. Banerjee and A. Roychoudhury. Automated Re-factoring of Android Apps to Enhance Energy-Efficiency. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 139–150, May 2016. doi: 10.1109/MobileSoft.2016.038.

[18] Titus Barik, Brittany Johnson, and Emerson Murphy-Hill. I heart hacker news: Expanding qualitative research findings by analyzing social news websites. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 882–885, Bergamo, Italy, 2015. ACM Press. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2803200.

[19] Daniel Bedard, R. Fowler, M. Lim, and Allan Porterfield. Powermon 2: Fine-grained, integrated power measurement. 2009.

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

[21] A. B. Bener, A. Miranskyy, and S. Raspudic. Deploying and provisioning green software. *IEEE Software*, 31(3):76–78, 2014.

[22] Dane Bertram, Amy Voida, Saul Greenberg, and Robert Walker. Communication, Collaboration, and Bugs: The Social Nature of Issue Tracking in Software Engineering. page 11, 2009.

[23] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 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, October 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

[24] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), November 2005. ISSN 0272-1732.

[25] Déaglán Connolly Bree and Mel Ó Cinnéide. Inheritance versus delegation: which is more energy efficient? In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*, pages 323–329. ACM, 2020. doi: 10.1145/3387940.3392192. URL https://doi.org/10.1145/3387940.3392192.

[26] Alexander Edward Ian Brownlee, Nathan Burles, and Jerry Swan. Search-Based Energy Optimization of Some Ubiquitous Algorithms. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):188–201, June 2017. ISSN 2471-285X. doi: 10.1109/TETCI.2017.2699193.

[27] Martin Burtscher, I. Zecena, and Ziliang Zong. Measuring gpu power with the k20 built-in sensor. In *GPGPU@ASPLOS*, 2014.

[28] Dimitrios Chasapis, Martin Schulz, Marc Casas, Eduard Ayguadé, Mateo Valero, Miquel Moretó, and Jesus Labarta. Runtime-Guided Mitigation of Manufacturing Variability in Power-Constrained Multi-Socket NUMA Nodes. In *Proc. of the 2016 International Conference on Supercomputing - ICS '16*. ACM Press, 2016.

[29] T. Chiba, T. Yoshimura, M. Horie, and H. Horii. Towards selecting best combination of sql-on-hadoop systems and jvms. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 245–252, 2018. doi: 10.1109/CLOUD.2018.00038.

[30] Shaiful Alam Chowdhury, Abram Hindle, Rick Kazman, Takumi Shuto, Ken Matsui, and Yasutaka Kamei. Greenbundle: an empirical study on the energy impact of bundled processing. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1107–1118. IEEE / ACM, 2019. doi: 10.1109/ICSE.2019.00114. URL https://doi.org/10.1109/ICSE.2019.00114.

[31] Henry Coles, Yong Qin, and Phillip Price. Comparing Server Energy Use and Efficiency Using Small Sample Sizes. Technical Report LBNL-6831E, 1163229, November 2014.

[32] Maxime Colmant, Romain Rouvoy, Mascha Kurpicz, Anita Sobe, Pascal Felber, and Lionel Seinturier. The next 700 CPU power models. *Journal of Systems and Software*, 144, 2018.

[33] Luis Corral, Anton B. Georgiev, Alberto Sillitti, and Giancarlo Succi. Method Reallocation to Reduce Energy Consumption: An Implementation in Android OS. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1213–1218, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2469-4. doi: 10.1145/2554850.2555064.

[34] John W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Method Approaches*. Sage Publications, Thousand Oaks, Calif, 2nd ed edition, 2003. ISBN 978-0-7619-2441-8 978-0-7619-2442-5.

[35] L. Cruz, R. Abreu, and J. Rouvignac. Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 205–206, May 2017. doi: 10.1109/MOBILESoft.2017.21.

[36] Luis Cruz and Rui Abreu. Performance-based guidelines for energy efficient mobile applications. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 46–57. IEEE, 2017. doi: 10.1109/MOBILESoft.2017.19. URL https://doi.org/10.1109/MOBILESoft.2017.19.

[37] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. 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, 2016. Association for Computing Machinery. ISBN 9781450343053. doi: 10.1145/2989081.2989088. URL https://doi.org/10.1145/2989081.2989088.

[38] Mohammed El Mehdi Diouri, Olivier Gluck, Laurent Lefevre, and Jean-Christophe Mignot. Your cluster is not power homogeneous: Take care when designing green schedulers! In *2013 International Green Computing Conference Proc.* IEEE, June 2013.

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

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

[41] Muhammad Fahad, Arsalan Shahid, Ravi Reddy Manumachu, and Alexey Lastovetsky. A comparative study of methods for measurement of energy of computing. *Energies*, 12 (11), 2019. ISSN 1996-1073. doi: 10.3390/en12112204. URL https://www.mdpi.com/1996-1073/12/11/2204.

[42] Benito Fernandes, Gustavo Pinto, and Fernando Castor. Assisting Non-Specialist Developers to Build Energy-Efficient Software. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 158–160, Buenos Aires, Argentina, May 2017. IEEE. ISBN 978-1-5386-1589-8. doi: 10.1109/ICSE-C.2017.133.

[43] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers. In *CCGRID 2020 - 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, Melbourne, Australia, May 2020. doi: 10.1109/CCGrid49817.2020.00-45. URL https://hal.inria.fr/hal-02470128.

[44] Alcides Fonseca, Rick Kazman, and Patricia Lago. A manifesto for energy-aware software. *IEEE Softw.*, 36(6):79–82, 2019. doi: 10.1109/MS.2019.2924498. URL https://doi.org/10.1109/MS.2019.2924498.

[45] Denae Ford, Titus Barik, Leslie Rand-Pickett, and Chris Parnin. The Tech-Talk Balance: What Technical Interviewers Expect from Technical Candidates. In *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 43–48, Buenos Aires, Argentina, May 2017. IEEE. ISBN 978-1-5386-4039-5. doi: 10.1109/CHASE.2017.8.

[46] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[47] Thomas Fritz and Gail C. Murphy. Determining relevancy: How software developers determine relevant information in feeds. In *Proceedings of the 2011 Annual Conference on Human Factors in Computing Systems - CHI '11*, page 1827, Vancouver, BC, Canada, 2011. ACM Press. ISBN 978-1-4503-0228-9. doi: 10.1145/1978942.1979206.

[48] Marion Gottschalk, Jan Jelschen, and Andreas Winter. Energy-Efficient Code by Refactoring. *Softwaretechnik-Trends*, 33(2):23–24, May 2013. ISSN 0720-8928. doi: 10.1007/s40568-013-0030-4.

[49] Miguel Guimarães, João Saraiva, and Orlando Belo. Some heuristic approaches for reducing energy consumption on database systems. *DBKDA 2016*, page 59, 2016.

[50] Ashish Gupta, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, Thirumalesh Bhat, and Syed Emran. Mining energy traces to aid in software development: An empirical case study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327749. doi: 10.1145/2652524.2652578. URL https://doi.org/10.1145/2652524.2652578.

[51] Sarra Habchi, Xavier Blanc, and Romain Rouvoy. On adopting linters to deal with performance concerns in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 6–16, Montpellier, France, 2018. ACM Press. ISBN 978-1-4503-5937-5. doi: 10.1145/3238147.3238197.

[52] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle. Energy profiles of java collections classes. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 225–236, May 2016. doi: 10.1145/2884781.2884869.

[53] Franz Heinrich, Alexandra Carpen-Amarie, Augustin Degomme, Sascha Hunold, Arnaud Legrand, Anne-Cécile Orgerie, and Martin Quinson. Predicting the Performance and the Power Consumption of MPI Applications With SimGrid. 2017.

[54] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. GreenMiner: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 12–21, Hyderabad, India, 2014. ACM Press. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597097.

[55] Ralph Hintemann and Simon Hinterholzer. Energy consumption of data centers worldwide. page 8.

[56] Jason M. Hirst, Jonathan R. Miller, Brent A. Kaplan, and Derek D. Reed. Watts up? pro ac power meter for automated energy recording. *Behavior Analysis in Practice*, 6(1):82–95, Jun 2013. ISSN 2196-8934. doi: 10.1007/BF03391795. URL https://doi.org/10.1007/BF03391795.

[57] Emanuele Iannone, Fabiano Pecorelli, Dario Di Nucci, Fabio Palomba, and Andrea De Lucia. Refactoring android-specific energy smells: A plugin for android studio. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 451–455. ACM, 2020. doi: 10.1145/3387904.3389298. URL https://doi.org/10.1145/3387904.3389298.

[58] Thomas Ilsche, Daniel Hackenberg, Stefan Graul, Robert Schöne, and Joseph Schuchart. Power measurements for compute nodes: Improving sampling rates, granularity and accuracy. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2015. doi: 10.1109/IGCC.2015.7393710.

[59] Yuichi Inadomi, Masatsugu Ueda, Masaaki Kondo, Ikuo Miyoshi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, and Keiichiro Fukazawa. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*. ACM Press, 2015.

[60] Jae-Jin Park, Jang-Eui Hong, and Sang-Ho Lee. Investigation for Software Power Consumption of Code Refactoring Techniques. In *SEKE*, 2014.

[61] Erik Jagroep, Jan Martijn E. M. van der Werf, Slinger Jansen, Miguel Ferreira, and Joost Visser. Profiling energy profilers. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, page 2198–2203, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450331968. doi: 10.1145/2695664.2695825. URL https://doi.org/10.1145/2695664.2695825.

[62] Erik Jagroep, Giuseppe Procaccianti, Jan Martijn van der Werf, Sjaak Brinkkemper, Leen Blom, and Rob van Vliet. Energy efficiency on the product roadmap: An empirical study across releases of a software product: Energy efficiency on the product roadmap. *Journal of Software: Evolution and Process*, 29(2):e1852, February 2017. ISSN 20477473. doi: 10.1002/smr.1852.

[63] Joakim v Kisroski, Hansfreid Block, John Beckett, Cloyce Spradling, Klaus-Dieter Lange, and Samuel Kounev. Variations in CPU Power Consumption. In *ICPE*. ACM, March 2016.

[64] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, San Francisco, CA, USA, May 2013. IEEE. ISBN 978-1-4673-3076-3 978-1-4673-3073-2. doi: 10.1109/ICSE.2013.6606613.

[65] Tomas Kalibera, Matthew Mole, Richard E. Jones, and Jan Vitek. A black-box approach to understanding concurrency in dacapo. In Gary T. Leavens and Matthew B. Dwyer, 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, 2012. doi: 10.1145/2384616.2384641. URL https://doi.org/10.1145/2384616.2384641.

[66] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004. ISBN 0321213351.

[67] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), March 2018. ISSN 2376-3639.

[68] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), March 2018. ISSN 2376-3639. doi: 10.1145/3177754. URL https://doi.org/10.1145/3177754.

[69] Foutse Khomh and S. Amirhossein Abtahizadeh. Understanding the impact of cloud patterns on performance and energy consumption. *J. Syst. Softw.*, 141:151–170, 2018. doi: 10.1016/j.jss.2018.03.063. URL https://doi.org/10.1016/j.jss.2018.03.063.

[70] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[71] Nupur Kothari and Arka Bhattacharya. Joulemeter: Virtual machine power measurement and management. *MSR Tech Report*, 2009.

[72] Mohit Kumar, Youhuizi Li, and Weisong Shi. Energy consumption in Java: An early experience. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, Orlando, FL, October 2017. IEEE. ISBN 978-1-5386-3470-7. doi: 10.1109/IGCC.2017.8323579.

[73] Mascha Kurpicz, Anne-Cécile Orgerie, and Anita Sobe. How much does a VM cost? Energy-proportional Accounting in VM-based Environments. In *PDP: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, page 8, Heraklion, Greece, February 2016. doi: 10.1109/PDP.2016.70. URL https://hal.inria.fr/hal-01276913.

[74] Sébastien Lafond and Johan Lilius. An Energy Consumption Model for an Embedded Java Virtual Machine. In Werner Grass, Bernhard Sick, and Klaus Waldschmidt, editors, *Architecture of Computing Systems - ARCS 2006*, volume 3894, pages 311–325. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-32765-3 978-3-540-32766-0. doi: 10.1007/11682127_22.

[75] James H. Laros, Phil Pokorny, and David DeBonis. Powerinsight - a commodity power measurement capability. In *2013 International Green Computing Conference Proceedings*, pages 1–6, 2013. doi: 10.1109/IGCC.2013.6604485.

[76] Thomas D. LaToza, Evelina Shabani, and Andre van der Hoek. A study of architectural decision practices. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 77–80, San Francisco, CA, USA, May 2013. IEEE. ISBN 978-1-4673-6290-0. doi: 10.1109/CHASE.2013.6614735.

[77] Michael LeBeane, Jee Ho Ryoo, Reena Panda, and Lizy Kurian John. Watt watcher: Fine-grained power estimation for emerging workloads. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 106–113, 2015. doi: 10.1109/SBAC-PAD.2015.26.

[78] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008. In Walter Binder, Vittorio Cortellessa, Anne Koziolek, Evgenia Smirni, and Meikel Poess, 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, 2017. doi: 10.1145/3030207.3030211. URL https://doi.org/10.1145/3030207.3030211.

[79] Peter Libič, Lubomír Bulej, Vojtěch Horky, and Petr Tůma. 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, 2014. Association for Computing Machinery. ISBN 9781450327336. doi: 10.1145/2568088.2568097. URL https://doi.org/10.1145/2568088.2568097.

[80] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 2–11, Hyderabad, India, 2014. ACM Press. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597085.

[81] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Multi-objective optimization of energy consumption of guis in android apps. *ACM Trans. Softw. Eng. Methodol.*, 27(3), September 2018. ISSN 1049-331X. doi: 10.1145/3241742. URL https://doi.org/10.1145/3241742.

[82] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[83] Kenan Liu, Gustavo Pinto, and Yu David Liu. Data-oriented characterization of application-level energy optimization. In *International Conference on Fundamental Approaches to Software Engineering*, pages 316–331. Springer, 2015.

[84] Mathias Longo, Ana Rodriguez, Cristian Mateos, and Alejandro Zunino. Reducing energy usage in resource-intensive Java-based scientific applications via micro-benchmark based code refactorings. *Computer Science and Information Systems*, 16(2):541–564, 2019. ISSN 1820-0214, 2406-1018. doi: 10.2298/CSIS180608009L.

[85] Y. Lyu, J. Gui, M. Wan, and W. G. J. Halfond. An empirical study of local database usage in android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 444–455, 2017. doi: 10.1109/ICSME.2017.75.

[86] Jens Malmodin and Dag Lundén. The energy and carbon footprint of the global ict and e and m sectors 2010–2015. *Sustainability*, 10(9), 2018. ISSN 2071-1050. doi: 10.3390/su10093027. URL https://www.mdpi.com/2071-1050/10/9/3027.

[87] Javier Mancebo, Coral Calero, and Felix Garcia. Does maintainability relate to the energy consumption of software? a case study. *Software Quality Journal*, 29(1):101–127, March 2021. ISSN 0963-9314, 1573-1367. doi: 10.1007/s11219-020-09536-9.

[88] Irene Manotas, Cagri Sahin, James Clause, Lori Pollock, and Kristina Winbladh. Investigating the impacts of web servers on web application energy usage. In *2013 2nd International Workshop on Green and Sustainable Software (GREENS)*, pages 16–23, San Francisco, CA, USA, May 2013. IEEE. ISBN 978-1-4673-6267-2. doi: 10.1109/GREENS.2013.6606417.

[89] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568297. URL https://doi.org/10.1145/2568225.2568297.

[90] Irene Manotas, Lori Pollock, and James Clause. SEEDS: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 503–514, Hyderabad, India, 2014. ACM Press. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568297.

[91] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, pages 237–248, Austin, Texas, 2016. ACM Press. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884810.

[92] Filippo Mantovani and Enrico Calore. Performance and power analysis of hpc workloads on heterogeneous multi-node clusters. *Journal of Low Power Electronics and Applications*, 8 (2), 2018. ISSN 2079-9268. doi: 10.3390/jlpea8020013. URL https://www.mdpi.com/2079-9268/8/2/13.

[93] Aniruddha Marathe, Yijia Zhang, Grayson Blanks, Nirmal Kumbhare, Ghaleb Abdulla, and Barry Rountree. An empirical survey of performance and energy efficiency variation on Intel processors. In *Proc. of the 5th International Workshop on Energy Efficient Supercomputing - E2SC'17*. ACM Press, 2017.

[94] David Margery, Emile Morel, Lucas Nussbaum, Olivier Richard, and Cyril Rohr. Resources Description, Selection, Reservation and Verification on a Large-scale Testbed. In *TRIDENTCOM - 9th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, May 2014.

[95] E. Masanet, Arman Shehabi, Nuoa Lei, Sarah J. Smith, and J. Koomey. Recalibrating global data center energy-use estimates. *Science*, 367:984 – 986, 2020.

[96] Hemant Kumar Mehta, Paul Harvey, Omer Rana, Rajkumar Buyya, and Blesson Varghese. Wattsapp: Power-aware container scheduling. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 79–90, 2020. doi: 10.1109/UCC48980.2020.00027.

[97] Sambit Kumar Mishra, Deepak Puthal, Bibhudatta Sahoo, Prem Prakash Jayaraman, Song Jun, Albert Y. Zomaya, and Rajiv Ranjan. Energy-efficient vm-placement in cloud data center. *Sustainable Computing: Informatics and Systems*, 20:48–55, 2018. ISSN 2210-5379. doi: https://doi.org/10.1016/j.suscom.2018.01.002. URL https://www.sciencedirect.com/science/article/pii/S2210537917302536.

[98] Iman Hemati Moghadam and Mel Ó Cinnéide. Code-Imp: A tool for automated search-based refactoring. In *Proceeding of the 4th Workshop on Refactoring Tools - WRT '11*, page 41, Waikiki, Honolulu, HI, USA, 2011. ACM Press. ISBN 978-1-4503-0579-2. doi: 10.1145/1984732.1984742.

[99] Rodrigo Morales, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. EARMO: an energy-aware refactoring approach for mobile apps. *IEEE Trans. Software Eng.*, 44(12):1176–1206, 2018. doi: 10.1109/TSE.2017.2757486. URL https://doi.org/10.1109/TSE.2017.2757486.

[100] Emanuel Moreira, Filipe F. Correia, and João Bispo. Overviewing the liveness of refactoring for energy efficiency. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, pages 211–212, Porto Portugal, March 2020. ACM. ISBN 978-1-4503-7507-8. doi: 10.1145/3397537.3397538.

[101] M. Morisio, P. Lago, N. Meyer, H. A. Müller, and G. Scanniello. 4th international workshop on green and sustainable software (greens 2015). In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 981–982, 2015.

[102] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Monitoring energy hotspots in software. *Automated Software Engineering*, 22(3):291–332, 2015.

[103] Adel Noureddine, Syed Islam, and Rabih Bashroush. Jolinar: Analysing the energy footprint of software applications (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 445–448, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2948706. URL https://doi.org/10.1145/2931037.2948706.

[104] H. Oi. Power-performance analysis of jvm implementations. In *ICIMU 2011 : Proceedings of the 5th international Conference on Information Technology Multimedia*, pages 1–7, 2011. doi: 10.1109/ICIMU.2011.6122743.

[105] D. G. Oliver, J. M. Serovich, and T. L. Mason. Constraints and Opportunities with Interview Transcription: Towards Reflection in Qualitative Research. *Social Forces*, 84(2): 1273–1289, December 2005. ISSN 0037-7732, 1534-7605. doi: 10.1353/sof.2006.0023.

[106] Mazliza Othman and Stephen Hailes. Power Conservation Strategy for Mobile Computers Using Load Sharing. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2(1):44–51, January 1998. ISSN 1559-1662. doi: 10.1145/584007.584011.

[107] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study. *ACM Transactions on Software Engineering and Methodology*, 25(3):1–53, August 2016. ISSN 1049-331X, 1557-7392. doi: 10.1145/2932631.

[108] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, Joel Penhoat, and Lionel Seinturier. 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, 2020. Association for Computing Machinery. ISBN 9781450369916. doi: 10.1145/3358960.3379142. URL https://doi.org/10.1145/3358960.3379142.

[109] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. On reducing the energy consumption of software: From hurdles to requirements. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375801. doi: 10.1145/3382494.3410678. URL https://doi.org/10.1145/3382494.3410678.

[110] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, and Joel Penhoat. Evaluating the impact of java virtual machines on energy consumption. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2021.

[111] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. Evaluating the energy consumption of java i/o apis. In *In Proceedings of the 37th International Conference on Software Maintenance and Evolution*, 2021.

[112] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. Tales from the code #1: The effective impact of code refactorings on software energy consumption. In *In Proceedings of the 16th International Conference on Software Technologies*, pages 34–46, 2021. ISBN 978-989-758-523-4.

[113] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. On the impact of code smells on the energy consumption of mobile applications. *Inf. Softw. Technol.*, 105:43–55, 2019. doi: 10.1016/j.infsof.2018.08.004. URL https://doi.org/10.1016/j.infsof.2018.08.004.

[114] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, May 2016. ISSN 1937-4194. doi: 10.1109/MS.2015.83.

[115] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. 2011. doi: 10.1145/2070562.2070567. URL https://doi.org/10.1145/2070562.2070567.

[116] Patton, Michael Quinn. *Qualitative Evaluation and Research Methods, 2nd Ed.* Number 0-8039-3779-2. Sage Publications, Inc, 2 edition, 1990.

[117] Rui Pereira, Tiago Carcao, Marco Couto, Jacome Cunha, Joao Paulo Fernandes, and Joao Saraiva. Helping Programmers Improve the Energy Efficiency of Source Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 238–240, Buenos Aires, Argentina, May 2017. IEEE. ISBN 978-1-5386-1589-8. doi: 10.1109/ICSE-C.2017.80.

[118] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? pages 256–267. ACM Press, 2017. ISBN 978-1-4503-5525-4. doi: 10.1145/3136014.3136031.

[119] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 22–31, Hyderabad, India, 2014. ACM Press. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597110.

[120] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '14*, pages 345–360, Portland, Oregon, USA, 2014. ACM Press. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660235.

[121] Gustavo Pinto, Francisco Soares-Neto, and Fernando Castor. Refactoring for Energy Efficiency: A Reflection on the State of the Art. In *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*, pages 29–35, Florence, Italy, May 2015. IEEE. ISBN 978-1-4673-7049-3. doi: 10.1109/GREENS.2015.12.

[122] Gustavo Pinto, Kenan Liu, Fernando Castor, and Yu David Liu. A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 20–31, Raleigh, NC, USA, October 2016. IEEE. ISBN 978-1-5090-3806-0. doi: 10.1109/ICSME.2016.34.

[123] G. Procaccianti, P. Lago, A. Vetrò, D. M. Fernández, and R. Wieringa. The green lab: Experimentation in software energy efficiency. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 941–942, 2015.

[124] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas

Würthinger, and Walter Binder. 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, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314637. URL https://doi.org/10.1145/3314221.3314637.

[125] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: benchmarking suite for parallel applications on the JVM. In *PLDI*, pages 31–47. ACM, 2019.

[126] Haris Ribic and Yu David Liu. AEQUITAS: Coordinated Energy Management Across Parallel Applications. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 4:1–4:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4361-9. doi: 10.1145/2925426.2926260.

[127] G. Rocha, F. Castor, and G. Pinto. Comprehending energy behaviors of java i/o apis. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019. doi: 10.1109/ESEM.2019.8870158.

[128] Ana Rodriguez. Reducing Energy Consumption of Resource-Intensive Scientific Mobile Applications via Code Refactoring. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 475–476, Buenos Aires, Argentina, May 2017. IEEE. ISBN 978-1-5386-1589-8. doi: 10.1109/ICSE-C.2017.33.

[129] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. Automatic microbenchmark generation to prevent dead code elimination and constant folding. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 132–143, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. doi: 10.1145/2970276.2970346. URL https://doi.org/10.1145/2970276.2970346.

[130] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14*, pages 1–10, Torino, Italy, 2014. ACM Press. ISBN 978-1-4503-2774-9. doi: 10.1145/2652524.2652538.

[131] Todd Sedano, Paul Ralph, and Cecile Peraire. Software Development Waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 130–140, Buenos Aires, May 2017. IEEE. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.20.

[132] Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Joseph P. White, Steven M. Gallo, Robert L. DeLeon, and Thomas R. FOPTurlani. Effect of meltdown and spectre patches on the performance of HPC applications. *CoRR*, abs/1801.04329, 2018.

[133] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Transactions on Software Engineering*, 45(9):877–897, September 2019. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2018.2810116.

[134] Stephen Romansky, Neil C Borle, Shaiful Chowdhury, Abram Hindle, and Russ Greiner. Deep Green: Modelling time-series of software energy consumption. In *ICSME*, 2017.

[135] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 120–131, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884833. URL https://doi.org/10.1145/2884781.2884833.

[136] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. *CoRR*, abs/1906.02243, 2019. URL http://arxiv.org/abs/1906.02243.

[137] The shift Project. Lean ICT - Towards Digital Sobriety. Technical report, March 2019.

[138] Kristin Fjola Tomasdottir, Mauricio Aniche, and Arie van Deursen. Why and how JavaScript developers use linters. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 578–589, Urbana, IL, October 2017. IEEE. ISBN 978-1-5386-2684-9. doi: 10.1109/ASE.2017.8115668.

[139] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180206. URL http://doi.acm.org/10.1145/3180155.3180206.

[140] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 2020. doi: 10.1109/TSE.2020.3007722.

[141] J.W. Tschanz, J.T. Kao, S.G. Narendra, R. Nair, D.A. Antoniadis, A.P. Chandrakasan, and V. De. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *IEEE Journal of Solid-State Circuits*, 37(11), November 2002. ISSN 0018-9200.

[142] Erik van der Kouwe, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Gernot Heiser. Benchmarking Crimes: An Emerging Threat in Systems Security. *CoRR*, abs/1801.02381, 2018.

[143] Georgios Varsamopoulos, Ayan Banerjee, and Sandeep K. S. Gupta. Energy Efficiency of Thermal-Aware Job Scheduling Algorithms under Various Cooling Models. In *Contemporary Computing*, volume 40. Springer, 2009.

[144] A. Vasan, A. Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah. Worth their watts? - an empirical study of datacenter servers. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, Jan 2010.

[145] Konstantina Vasileiou, Julie Barnett, Susan Thorpe, and Terry Young. Characterising and justifying sample size sufficiency in interview-based studies: Systematic analysis of qualitative health research over a 15-year period. *BMC Medical Research Methodology*, 18 (1), December 2018. ISSN 1471-2288. doi: 10.1186/s12874-018-0594-7.

[146] R. Verdecchia, G. Procaccianti, I. Malavolta, P. Lago, and J. Koedijk. Estimating energy impact of software releases and deployment strategies: The kpmg case study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 257–266, 2017. doi: 10.1109/ESEM.2017.39.

[147] W G P Silva, Lisane Brisolara, Ulisses Brisolara Corrêa, and Luigi Carro. Evaluation of the impact of code refactoring on embedded software efficiency. *Unpublished*, 2010. doi: 10.13140/2.1.1481.8249.

[148] Helmut R. Wagner. The Discovery of Grounded Theory: Strategies for Qualitative Research. By Barney G. Glaser and Anselm L. Strauss. Chicago: Aldine Publishing Company, 1967. 271 pp. . *Social Forces*, 46(4):555–555, 06 1968. ISSN 0037-7732. doi: 10.1093/sf/46.4.555. URL https://doi.org/10.1093/sf/46.4.555.

[149] Yewan Wang, David Nörtershäuser, Stéphane Le Masson, and Jean-Marc Menaud. Experimental Characterization of Variation in Power Consumption for Processors of Different generations.

[150] Yewan Wang, David Nörtershäuser, Stéphane Le Masson, and Jean-Marc Menaud. Potential effects on server power metering and modeling. *Wireless Networks*, November 2018. ISSN 1022-0038, 1572-8196. doi: 10.1007/s11276-018-1882-1.

[151] B. Whitehead, D. Andrews, and Amip Shah. The life cycle assessment of a uk data centre. *The International Journal of Life Cycle Assessment*, 20:332–349, 2015.

[152] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Unrestricted and safe dynamic code evolution for java. *Sci. Comput. Program.*, 78(5):481–498, 2013.