Università degli Studi di Milano Bicocca

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di Laurea Magistrale in Informatica

Green smell identification for software energy efficiency optimization

Relatore: Prof.ssa Francesca Arcelli Fontana

Co-relatore: Dott. Marco Bessi

Tesi di Laurea Magistrale di:

Corrado Ballabio

Matricola 764452

Anno Accademico 2016-2017

Table of contents

List of figures

List of tables

Chapter 1

Introduction

The role that Information and Communication Technologies (ICT - or just IT) have on everyday life is becoming more central every year passing by and is probably one of the most transformative developments of the last several decades, resulting in a growth of software-intensive systems use and production. These software systems have the ability to make our lives better, but on the other hand they also contribute to the damaging of the environment. The creation of always more performing (and thus more electricity-hungry) systems makes the total amount of power request no longer negligible, so that the polluting process that is enabled for their maintenance has a significant impact on global stability.

This chapter provides a brief introduction on the motivation behind the study of Green IT: through the analysis of data related to the power consumption of IT infrastructures and the relative consequences, it is explained why an optimization of the energy efficiency is required. It follows an overview of the thesis that illustrates the content of each chapter and the aim of this work.

1.1 ICT global impact

According the trends in IT systems production, all the three main categories of ICT (i.e. end-user devices, telecommunication networks and data centers) are experiencing a rapid growth that is expected to continue in the next future [30]. In particular, data center sector is projected to have the bigger increase of all with a rate of 7.1% per year, representing thus a significant example to analyze for understanding and foresee IT-related issues. Even statistics shows how the rise of data center power consumption slowed down in last years mainly due to the economic global crisis and the expansion of the virtualization approach [6, 28], electricity used for data center maintenance grew at a pace of over 50% a year in the recent

past [57]. In 2014 this quantity represented the 1.5% of global electricity use –reaching 2.2% in the U.S only– with an estimated consume between 203.4 and 271.8 TWh per year. In the only 6 years between 2007 and 2012, ICT electricity consumption increased by more than 200 TWh on a global scale, with a growing rate of 6.6% per year, going from 4 to 4.7% of the world total (Figure 1.1). This trend led to two main consequences: a raise in maintenance

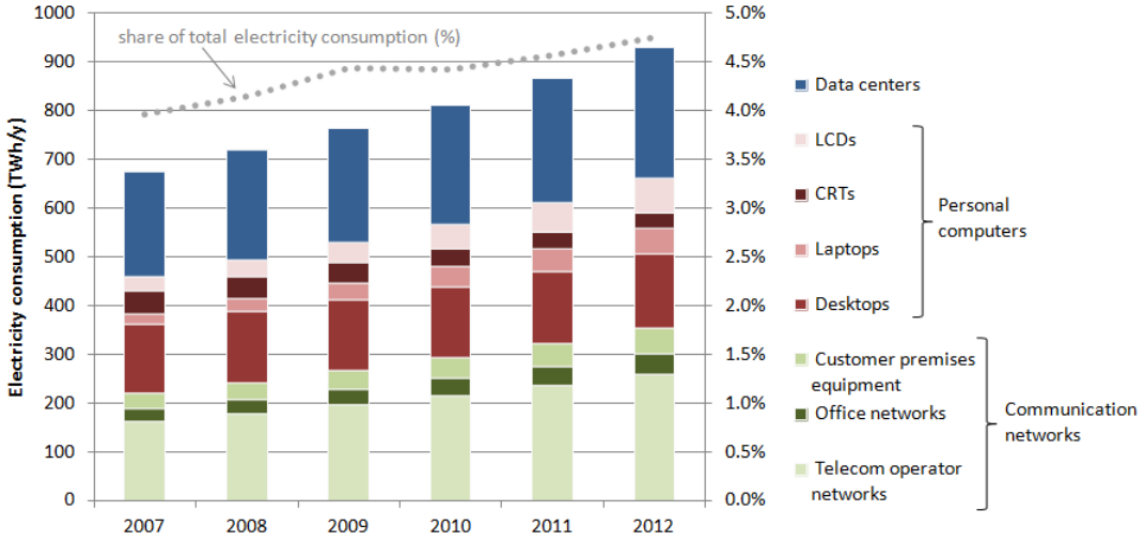


Fig. 1.1 Worldwide use phase electricity consumption of communication networks, personal computers and data centers.

cost of the infrastructures, and a consequential increment in Carbon Dioxide (CO_2) emission due to the industrial process that provides energy. A typical facility hosting a large data center requires between $10 million and $25 million to operate [54]. As the report issued by the Global e-Sustainability Initiative shows [30], the greenhouse gas emissions of the ICT sector are projected to rise to 1.3 $GtCO_2e$ (gigatonnes of equivalent carbon dioxide) by 2020, accounting for 2.3% on global scale, and for this reasons Data Centers expansion recently started to be accounted as one of main threats for the environment [55]. Even Greenpeace once reported that the rapid adoption of Cloud Computing technologies would have had as effect an increase in the demand for data centers in the next future [18], making them a focal point for the environment preservation community.

It is clear that an improvement in energy consumptions can lead to several benefits both on the environment and on an economic side. All IT companies should prioritize the research of techniques and practices that could make IT system more efficient and Eco-friendly.

1.2 Data Center consumption analysis

For a better understanding of the motivations behind the Green IT, energy consumption analysis of a sample data center facility is taken as example. As already stated, data centers represents the most rapid developing sector of ICT, so their analysis can represent a more relevant case.

The analysis of energy consumption in data centers has been well studied [3, 42], since the management of their behavior and their eventual improvement represents a crucial point in the productivity and economic spending in big IT companies. Energy Efficiency [41] is one of the most referred metrics when it comes to IT systems improvement, it refers to the practice of using less power while producing the same amount of services, or in other terms to the following ratio:

$$\text{Energy Efficiency} = \frac{\text{output of a process}}{\text{energy input to that process}}$$

As can be deduced from the definition, greater ratio of energy efficiency reflect a better utilization of the available resources. This means that *Energy Efficiency* plays a central role in consumption and maintenance cost. Several works [2, 42, 12, 35] focus the attention on server energy and its efficiency, identifying the areas in which the usage is flawed and could be improved, monitoring the CPU utilization levels from thousands of servers through several months of study. They provide a view on how power usage is partitioned and its distribution over time. Figure 1.2 splits energy requirements for each main component of a sample server. According to this data computational operations consist in slightly more than the half of the total consumption, while the remaining amount is drained for the cooling process of the system and other operations. The direct consequence is that 42% of the maintenance cost in a facility are associated with hardware, software, uninterrupted power supplies, and networking, and 58% of the expenses is due to heating and air conditioning [54]. This is justified by the fact that whereas the cost of hardware production has only slightly grown in the last years, the cost of power and cooling management has grown over four times [24].

Power consumption distribution over time reported in Figure 1.3 tells that servers are rarely completely idle and seldom operate at full capacity. In detail, they spend the majority of their time between 10% and 50% of their maximum performance levels. This means that even during periods of low service demand, servers are unlikely to be fully idle, leading to a considerable waste of capital due an over-dimensioning of the service. On the contrary, servers that operate close to 100% capabilities usually have difficulties in meeting throughput and latency service-level agreements because of hardware or software faults. In Figure 1.4 server power usage is shown in relation to energy efficiency. Better values of efficiency correspond to peak of usage, and conversely values diminish with usage decrease. In the area

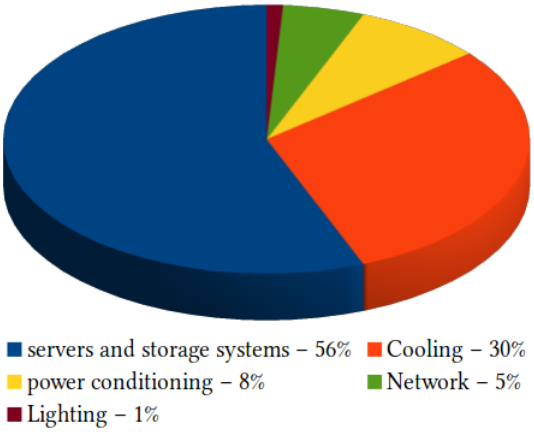


Fig. 1.2 Energy requirements of server components

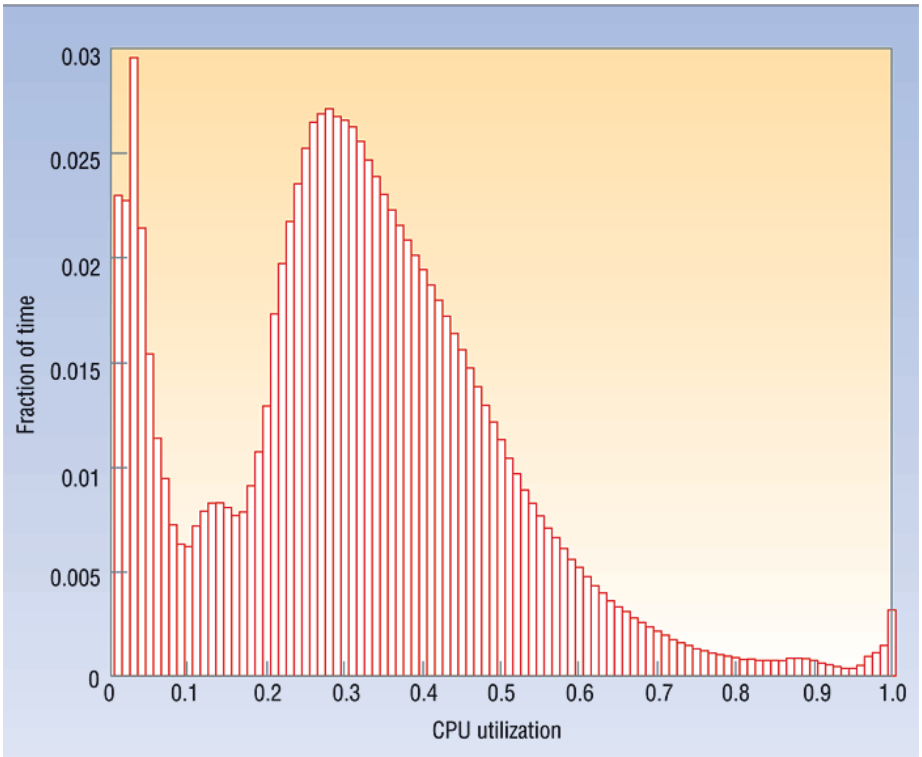


Fig. 1.3 Average CPU utilization over time in typical servers.

server spend more time in –that corresponds to the segment between 20%-30% of utilization– efficiency is merely less then an half of the peak performance: about half of the power is then spent by the server when is virtually doing nothing. This trend results in poor power management behavior.

The amount of power consumed by data centers does not reflects the amount of work the system is doing. This is because the principal design objective for most IT systems to date

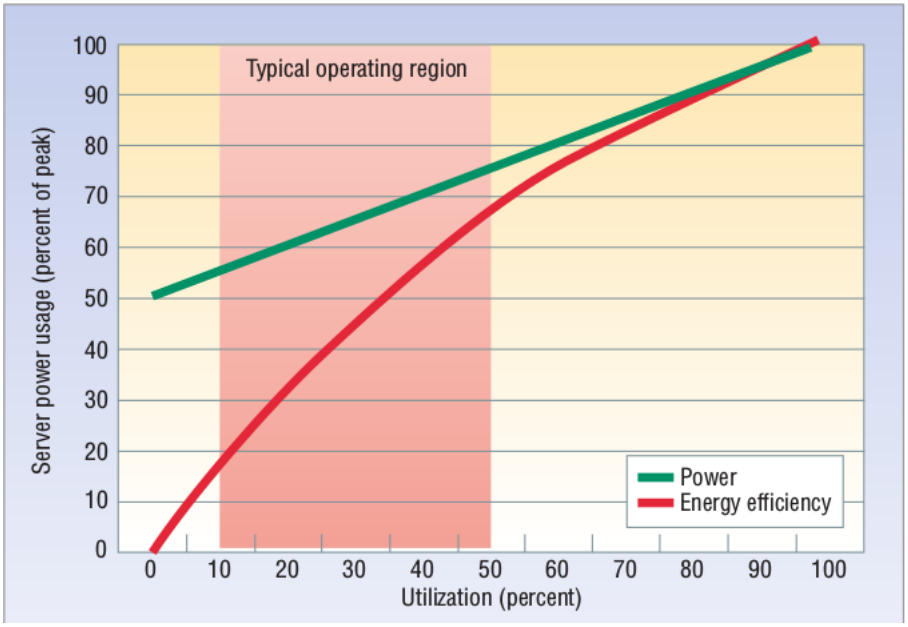


Fig. 1.4 Relation between power usage and energy efficiency in servers.

has been to maximize computational performance –or perhaps, performance at a given price point– with very little consideration given to energy use. Most enterprises prefer to adopt cost-effective solutions and they focus on the trade-off between cost and quality, while neglecting software energy efficiency. Especially in dedicated field like high-performance computing (HPC), the quest for achieving maximum performance in terms of number of floating-point operations per second (flops) has led to neglect the power consumption issue, leading to extremely energy inefficient results. Initiatives like Green500 [27, 21] try to put the stress on HPC related consumption and succeeded in proving that the performance-at-any-cost paradigm is no longer feasible.

All these studies give an idea on how much data centers, and IT computation in general terms, are still inefficient and despite all the research works and the effort put by the scientific community, the margin of improvement is still substantial.

1.3 Thesis outline

This study focuses on the Green IT issue, addressing the following objectives:

1. identification of a set of energetically inefficient Java code patterns, called *"energy code smells"* [62] or, adopting a new term for indicating the positive environmental impact they bring, *"green smells"*. Starting from an initial selection of known programming

bugs, tests will be performed in order to identify those who represent energy hotspots, that then have a negative impact on the total power consumption in Java applications;

2. analysis of possible relationships between power consumption and other variables such as execution time, memory usage and bytecode instructions. Additional tests will identify the causes behind the energy waste caused by green smells;

3. creation of an index that assesses the energetic efficiency of given applications. This metric, named *greenability index*, will be implemented using the Assessment Model of CAST [1] Application Intelligence Platform (AIP), through the definition of custom quality rules with the previously detected green smells. The name greenability is intended to reflect the quality of being environmental-friendly for given applications.

In conclusion, the aim of this thesis is to answer to two research questions:

RQ1: which of the patterns under test represent an energy hotspot (i.e. which are green smells)?

RQ2: which are the main variables between execution time, resource usage and bytecode instruction, that are responsible of the increase in energy consume?

The rest of the thesis is organized as follows:

- *Chapter 2* presents the previous works related to Green IT studies, like improvements in energy efficiency, the definition of energy code smells, and different software energy measurement techniques.

- *Chapter 3* introduces preliminary informations useful for the works. A description of the main CAST company areas and its index related to energy consumption is given. Then an overview of jRAPL [32], the adopted energy measurement library, is presented.

- *Chapter 4* presents the early stages of the work. It describes the process of identification of the initial set of patterns under test, and then defines each identified bug.

- *Chapter 5* explains the setup of the performed tests and discuss the corresponding results. This analysis will define the set of green smells. Afterwards other tests for defining the relationships between energy spending, execution time and resources usage will be preformed and discussed too. A final analysis on bytecode operations concludes the chapter.

[1] https://www.castsoftware.com/

- *Chapter 6* illustrates the implementation of the detected green smells into Quality Rules of the CAST Assessment Model. This transformation produces a methodology that is able to assess the level of energy efficiency of given applications.
- *Chapter 7* gives the conclusion of the works, giving an input to possible future works.

Chapter 2

Related Work

Over the years, the use and the capabilities of IT have grown constantly making our life easier and helping our jobs. However, alongside with several benefits, IT has been contributing to environmental issues, becoming one of the main factors in CO_2 and other greenhouse gases emission. The Green IT research tries to face the problem by minimizing or eliminating where possible the environmental impact of IT and helps creating a more sustainable environment.

"(Green IT) is the study and practice of designing, manufacturing, using, and disposing of computers, servers, and associated subsystems - such as monitors, printers, storage devices, and networking and communications systems - efficiently and effectively with minimal or no impact on the environment." [38]

In general, the denomination Green IT may refers to several different fields:

- *IT as an enabler of green governance:* the measuring and monitoring of green parameters related to all business processes.
- *Eco-compatible management of IT products lifecycle:* research for the reduction of the environmental impact that IT components have, starting from the early stages to the disposal.
- *Energy efficiency of IT:* the designing of energy-efficient IT architectures and the consequent effects on energy consumption.

The first point is related to the laws and regulation that government or competent bodies apply in order to promote a more eco-friendly policy. This side does not belong to the interest of this work, so it will not be discussed here. The following section will introduce the works related to the other two sector of the Green IT.

2.1 Green IT works

Even if energy spending is not a central issue during the development process, it is gaining more attention every year passing. This trend is confirmed by the amount of Energy consumption-related questions across the most popular Q&A websites, which experienced a near-linear growth in the last 5 years, suggesting that it is becoming an always more central issue among programmers community [46]. Taking count of the numerosity of question, their respective answers, and the quality of them, the energy efficiency issue is two times more popular when compared to the average value of all the other topics.

There are two main aspects important in making software greener: on one side activities that make the production process more eco-friendly, and on the other the methodologies to generate a greener product itself. In ISO/TS 14067 [15] and in [58, 25] a Green Software Development Model is re-designed, including a set of best practices and eco-friendly methodologies in every phase of the software development lifecycle (SDLC), supporting the development of software systems in an environment friendly way. This process splits the several phases that occurs during the creation of IT product, starting from the requirements gathering, prototyping, implementation, the testing, deployment, the maintenance and eventually the disposal. Every step of this sequence has a margin of improvement from a sustainable point of view, starting from a reduction of material resource waste, like preferring re-usable prototyping instead of throwaway ones or avoid disposable media like CDs or DVDs in favor of electronic format documents. Surprising savings in power consumption can be achieved regulating the activity of devices in prolongated stand-by mode: printers can consume up to two thirds of their energy in idle state, while only the remaining one third is spent in active mode [61].

A more interesting approach in Green IT, as already said, is the one related to the improvement of energy efficiency. A better use of the available computing resources is the key in a smart power management.

"A dual strategy is needed to solve the energy problem: maximize energy efficiency and decrease energy use; develop new sources of clean energy. The former will remain the lowest hanging fruit for the next few decades." –Steve Chu, former director of the Lawrence Berkeley National Laboratory [10]

The bigger sector in energy efficiency research is represented by the hardware-related improvement of computational performance. The green efficiency idea started in 1992 when the US Environmental Protection Agency (EPA) introduced ENERGY STAR [5], a voluntary labeling approach to recognize electronic equipment's energy-efficiency characteristics.

It was designed to identify and promote energy-efficient products and make it easier for consumers to save money and protect the environment. Even if by its definition Green IT studies try to address the environmental problem through different approaches, the vast majority of the effort is put on hardware-related improvements. Enhancement of computing components such as the extension of battery lives or the boosting of processor performances reflect the bigger sector in the industry, but in recent years a different approach to the problem is gaining popularity.

2.2 Green software

In professor Koomey work [29], analyzing the performance of various software systems over the years (Figure 2.1) he formulates the so called *Koomey Law*. This formula establishes a relationship between performance and power usage, stating that the ratio between the number of computations per Joule and the energy dissipated doubles approximately every two and a half years, accordingly to a trend that has been remarkably stable over the years. On the other hand energy consumption of software systems increases linearly, defining a trend that is perfectly summarized by swiss computer scientist Niklaus Wirth: *"Software is getting slower more rapidly than hardware is getting faster"* [52]. In other words, while the performance of computations increase year after year due to the high confidence in constant hardware improvements, the number of computations needed to fulfill a task increases even more, following the *software bloat* [34] pattern. It becomes evident that a bigger effort needs to be made in order to let the software side keep the pace of hardware impetuous development.

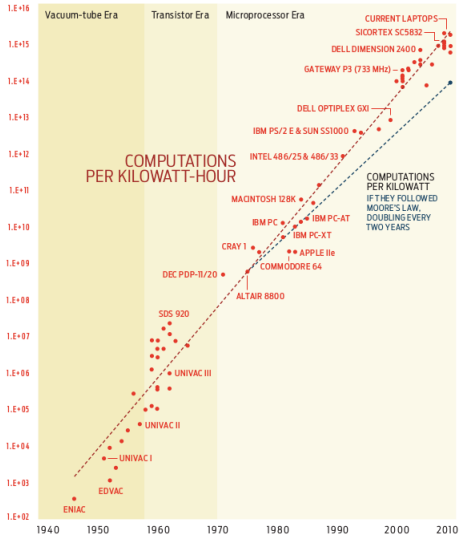


Fig. 2.1 Number of computations per Joule of energy over the years.

When it comes to IT power consumption, what is not immediate to grasp is that hardware behavior is strictly affected by the underlying software application indeed. Focusing on the motivations that can influence power consumption during computation, it can be affirmed that software itself (algorithm choices, code style and its efficiency...) can influence energy requirements, and it is indirectly responsible of its consumption. The basic idea is that inefficient software choices have a sort of propagation effect that leads to inefficient behavior in hardware and thus an improper use of power supplies. For each Watt that the processor spends to elaborate information, the total energy consumed by the system can be multiple times higher, due to over-work of drivers, memory, cooling system, back-up batteries and all the other auxiliary components. Thus, the benefits obtained by optimizing the energy consumed for computation are amplified by the above IT layers and have a great impact on the total consumption. Computational efficiency, data efficiency, and context-aware methods can all contribute into creating applications that are power-aware.

This causal relationship between software and the inducted hardware energy consumption has been proved [61, 48, 50], but a common yet overly simplistic belief is that the main factor in energy consumption is the run time of applications. Accordingly to the *race-to-idle* approach [1, 26], faster programs will theoretically consume less energy because the system can reach the idle state in faster time. Unfortunately it is quite hard to generalize the relationship between performance and energy consumption, especially for concurrent systems [45]. The nature of the problem to be solved, the technique used to manage concurrent execution, the CPU clock frequency and the JVM implementation are some of the factors that can contribute in making the trade-off less obvious. Faster is not a synonym for greener, that is why Green Software studies investigate the deeper reasons that make software energy inefficient.

Different aspects can affect the energy consumption of the underlying hardware infrastructure of a software product. Verdecchia et al. [60] performed an empirical assessment in which they compare the impact that different deployment strategies and releases of an industrial software product have on power requirements. These several releases implements the same functional requirements in different ways, in order to maintain the same use case scenarios during all the phases of the test. In conclusion of this experimental test, results show that deployment strategies do have a significant impact on energy consumption of a software product, with software releases variable giving a contribute in this evaluation. In addition, the findings also indicate how the power variation of underlying hardware might be negligible when optimizing energy efficiency at software level, since the most relevant factor is time variation (Energy is proportional to time and power consumption). Furthermore, data

does not show any absolute optimal combination for releases and deployment strategies, as the interaction of these two factors has to be considered according to the situation.

In his work Bessi [4] identifies a subset of Java pure functions as energy inefficient. A method is considered *pure* if it satisfies two properties: it is side-effects free, that is its execution does not generate any other effect beside the original result, and it is deterministic, that is its output depends only from the argument passed. It then implements an automatic detection and energy efficiency improvement of this subset of functions with a memoization-based approach that does not require a refactoring of code. Final results show that the approach provides a significant energy saving of over 86% of the overall value.

Multiple studies by Pinto et al. focus on the impact that different programming choices can have on the energy usage and performance. In [44] was performed an analysis on the main methods of 16 types of commonly used collection in the Java language, grouped in List, Set and Map. According to the operation implemented –insertion, removal or traversation–, or other "tuning knobs"such as the initial capacity and load factor, the results shows how energy consumption is not equal on every different configuration. In [47] they investigate how concurrent programming practices may have impact on energy consumption, finding that main factors are the choice of thread management constructs, the number of threads, the granularity of tasks, the size of the data, and the nature of data access. Bunse et al. [7] analyze the energy consumptions of different sorting algorithms. Since this class of algorithms is essential for managing data, results show that resources management plays a central role in power usage: in-place algorithms like insertion sort, even if less performing in specific settings, are more energy efficient than others.

The relation between energy consumption and cloud software architectural design of applications is studied in [49, 51]. An analysis of the state-of-the-art of cloud computing efficiency was first made, in order to define a set of typical techniques. These techniques was then codified in the form of *Green Architectural Tactics*, to support the design and development of cloud-based energy efficient software, and their architectural impact on power efficiency was analyzed in terms of assumptions and tradeoffs. Each of the resulting tactics is then described in terms of a motivation, a brief description, the constraints for its application, an example, and its dependencies with other tactics in order to be applied.

For what concern the interaction with database system, an empirical study on ORM (Object-Relational Mapping) [22] compares energy efficiency of different frameworks. The evaluation tests three accesses to SQL database using plain SQL directly from source code, Propel, a pure ORM framework, and TinyQueries, a "no-ORM", according to two additional factors like the database size and the kind of query performed (Create, Read, Update, Delete). Final results show how, even if ORMs introduce several benefits especially in

terms of code maintainability, flexibility and readability when compared to a plain SQL approach, it introduces other drawbacks in power consumption, leading to a trade-off situation. Measurements of Propel usage results in an increase of 70% in execution time, incrementing remarkably the energy usage. While best results are achieved in the test case that used plain SQL, a better balance in the trade-off is gained using TinyQueries, that joins the maintainability effects of ORMs approaches and energy efficiency of code-level queries.

2.2.1 Energy code smells

Taking inspiration from the Fowler and Beck definition of code smells [14] referring to bad code structures which are difficult to understand, read and maintain, the term *energy code smell* indicates energy hotspots, software-intrinsic elements or properties that have a measurable and significant impact on energy consumption, at any level of abstraction of the system architecture.

Related studies by Gottschalk et al. [23, 17, 16] operate towards energy savings on mobile phones. Smart energy management in this particular field assumes crucial importance: bad energy usage in mobile devices such as smartphones or laptops directly leads to a reduction in battery lifetime, such that it represents one of the biggest complains in users negative reviews [63]. Gottschalk identifies a set of six energy code smells that cause an excessive consume of battery resources on android smartphones and defines an automatic detection and reengineering approach for their resolution. Final results shows that the refactoring of those smells can lead to energy savings up to 56.6%.

Another relevant study from Vetrò et al. [62], investigates on energy code smells in C language is performed. They choose this specific programming language because their aim is to investigate power consumption in embedded systems, due to the fact that it acquires a peculiar importance since most embedded systems are battery-powered. The work leads to the identification of a set of code patterns and the performing of an experiment scenario on an embedded system. Each smell is executed in two versions: the first one contains a code issue that could negatively impact power consumption, the other one is refactored removing the issue. In addition to power consumption measurement, also execution time is monitored in order to investigate whether there is any correlation between these two dimensions. Final results show that energy code smells does not introduce any performance decrease, and the absence of a correlation between energy inefficiency and execution time.

2.2.2 Energy spending factors

As already mentioned in the previous section, a very common misconception when analyzing energy consumption in software is that the so called *race to idle* approach has a positive impact on the power consumption. This approach tells that an execution time reduction in a program brings the same amount of energy saving. However, accordingly to the Energy equation (E = power $\times$ time), indicates that the power variable of the equation, which cannot be assumed as a constant, also has an impact on the energy. The conclusion on the argument in literature are in disagreement, some supporting the energy-time relation [64], but most of them other discarding it [59, 62, 47, 31]. In addition to the energy-time relationship, in [43] allocated resources are taken in consideration as well. Final results shows that the relation between energy consumption and peak memory usage is almost non-existent, but despite it a possible influence could be done by *how* the memory is managed.

2.3 Energy Metrics

From a pure physical point of view, the energy consumed by a computational environment to complete a set of tasks can be written as

$$\mathrm{E} = \int_{t_{start}}^{t_{end}} P(t)dt \text{ [36]}$$

where t_{start} and t_{end} are starting and ending times of the set of tasks P(t) is the function of the instant power with respect to time, as defined

$$\mathrm{P}(t) = \sum_{i \in res} P_i \alpha_i(t)$$

Power consumption is then simply the sum of instant consumption of each resource; α_i; is the function of the utilization factor for the i-th resource at time t and P(i) the peak power consumption of the i-th resource.

Existing energy profiling approaches can be divided in two categories depending on their design pattern: software-based or hardware-based.

2.3.1 Software-based approaches

In their works, Capra et al. [9, 8] define firstly an energy measurement metric, then an energy efficiency one using a logical point of view . Energy measurement is defined through the logical depth definition (the smallest number of elementary logical operations required to perform the computation that produces the desired string of bits) and the Margolus-Levitin theorem (that defines the minimum commutation energy required by a system to operate at a given frequency): the theoretical lower bound of the energy consumption of a software

application is then: $EC_{min}(f) = E_{min}(f) \cdot L_d \cdot T_{d-min}$. E_{min} is the minimum energy required by a single bit status commutation, L_d is the logical depth of the required output and T_{d-min} is the most efficient way of representing the problem for the desired output, that is the minimum thermodynamic depth. On the other hand energy efficiency of an application i is defined as $EE_i = 1 - SE_{iNORM}$. SE_{iNORM} is the normalized value of all the S_i of applications belonging to the same functional area, with S_i as the power absorbed by the system while running the application i minus the power absorbed in idle.

The E-Surgeon approach [39] includes a 2-layer architecture that consists of PowerApi, a system monitoring library, and Jalen, a process-level monitoring agent. The proposed tool estimates the power consumption of applications directly from the source code, analyzing information gathered from both hardware devices through the operating system and from software through bytecode instrumentation. These two parts cooperate to provide accurate energy information at runtime. Even if E-Surgeon figures as a valid tool for detecting eventual energy hotspots in software execution, it introduces a not negligible overhead that in some cases can exceed 56%, that limits its usage in fine-grain measurements.

Another approach [36] uses only the total energy consumption and some information available at runtime to the scheduler like the execution time, the number of cores and the number of machine, and then builds an energy model. Applying linear regression, it can automatically extract values for the single features of the computational environment like the power absorbed by the infrastructure P_{infr}, the power absorbed by every single machine P_m and the difference consumption between the idle state and the active state $P_{\delta c}$,besides the prediction of peak power absorption and the estimation of energy consumption of a system executing concurrent parallel tasks with high level of accuracy.

Mukhanov introduces Alea [37], a probabilistic approach for estimating power and energy consumption at a fine-grained level. In particular, it proposes two different models: a *constant-power probabilistic model* and a *variation-aware probabilistic model*. The first one approximates the power consumption of coarse-grained code regions, assuming that the consumption value remains constant between two analysis intervals. Consequently the precision of this approximation heavily depends on the minimum interval allowed between the reading instants. The second one instead improves the estimation by identifying high-frequency variations at the fine-grained level, it detects power variation over intervals significantly shorter than the minimum feasible power-sensing interval. Alea achieves good level of accuracy of energy estimation, with worst-case errors under 2% for coarse-grained code blocks and 6% for fine-grained ones.

2.3.2 Hardware-based approaches

The hardware-based energy profiling approach on the other hand utilizes external hardware equipment such as power meter or multi-meter, to obtain voltage and current readings for estimating the power consumed by a computing system or some of its components. This specific instrumentation is specifically conceived for energy measurement, so the results using this approach are highly accurate. For this specific kind of analysis, multimeters and ammeters can provide precise measurements, but it is usually preferred to build custom monitoring systems that can be adapted to particular system configuration. This is preferred because this multi-purpose instruments are conceived to monitor the amount of electric current that flows inside a system in Ampere. The market has several tools for this task, but they are usually intended for the monitoring of domestic appliances, so they may not be suitable for software energy analysis.

In [13] the authors propose and designed a tool called PowerScope. This device uses a digital multimeter to sample both the energy consumed through the external power input and the activities of the profiling system. PowerScope maps the energy consumption to program structure, so that it can determine the energy consumed by a specific process, or even by different procedures within the process. The results are then analyzed in a second moment, generating an energy profile from the sampled data. This phase occurs later when off-line in order to avoid a profiling overhead that may interfere with the monitoring process.

When it comes to build a custom measurement kit for analysis purposes, different works adopted several approaches. For its work Bessi [4] assembled a custom scheme that consisted in several tools. The current absorbed by the system was monitored by an ammeter clamp. The collected data was then processed by a Data Acquisition Board that was interfaced via a USB with a different server from the one used for the test, in order to avoid overhead, where final analysis was performed.

The device used in the research work by Vetrò et al. [62] was a Waspmote V1.1[1], a device that is essentially is a motherboard with connectors to plug in other elements such as sensors and other wireless modules that has no operating system: programs are directly loaded on its FLASH memory. This kit was optimal because it provided a “clean”environment in which the noise was minimized due to the fact that the only executed code was the test workload, and it also provided a specific sensor for the execution time measurement.

In [60], it was used a *Wattsup PRO*[2], a power meter for measuring the overall consumption of the machine. This tool provides a USB interface for exporting the data and high precision

[1] http://www.libelium.com/development-v11/

[2] https://www.powermeterstore.com/p1206/watts_up_pro.php

of the measurement: it produces the total Watts, Voltage and Current consumption with relative timestamp for the whole duration of the analysis.

2.3.3 Comparison of the approaches

Both the approaches seems to have their advantages and disadvantages, depending on the situation they are intended to be used in [40]. An hardware-based solution provides more accurate results and do not require any interaction with the system under analysis, as it operates out of it through an external hardware. The downside is that it can be highly expensive due to hardware investment, labor-intensive and non-scalable. Results have a coarse granularity since it only measures the overall consumption. Additionally accuracy may decrease if the measurement is performed on device architectures different from the one they are specifically designed for.

A software-based approach gives the user different levels of granularity to choose from, such as process, thread, function or even line level, maximizing the power tracking resources. However it provides estimated of energy consumption based on gathered information through a power model. This approach may not always guarantee perfect levels of accuracy, though certainly high. There may also occur a negligible overhead due to the additional power required during the power logging phase. Lastly, in some cases software availability could also be restricted to specific CPU architectures or OS, due to limited support of available APIs.

2.4 Conclusions

Previous studies mainly focus in assessing the energetic impact that software may have on embedded systems or mobile devices. On the other hand, this work differs from the previous ones because its aim is to detect a consistent number of energy hotspots in general purpose Java applications and define a set of new *green smells*. Moreover, currently in literature the definition of a methodology to quantify software energy inefficiency is missing. Through the definition of an indec that measures the *greenability* of applications, this work will provide a solution.

Chapter 3

Background

This chapter provides preliminary informations that needs to be introduced before the actual work is presented.

Firstly, a brief presentation of CAST company introduces its Application Intelligence Platform, Health Factors and Green IT index. A discussion about energy measurement methodology follows, and illustrates the features of jRAPL library, that will be used in this work. Finally, some Automatic Static Tools tools that will be used for the detection of the patterns to test are presented.

3.1 CAST

CAST[1] is an independent software vendor that is a pioneer and world leader in Software Analysis and Measurement (SAM). It was founded in Paris in 1990, its headquarters are located in New York and has a presence in the Americas, Europe, Middle East, Africa and India, serving IT-intensive enterprises and public sector institutions on worldwide scale. CAST introduces fact-based transparency into software asset management, application development, maintenance and sourcing to transform it into a management discipline. Its mission is to provide a standard unit of measure for those who build, buy or sell software.

The analysis that CAST performs on tested applications has a double scope: it inspect both at code level, detecting readability, maintenability, cyclomatic complexity, and polymorphism related issues; both at system level. This additional feature permits to expose resiliency, stability, security, performance and data integrity problems, that the former is not able to detect.

[1] http://www.castsoftware.com/

"Bad architectural practices account for only 8% of programming mistakes but lead to 90% of production issues."[2]

CAST provides a software analysis tool that helps to determine the overall quality of applications under test, and to identify some of the root causes of current resilience issues, as well as any risk of possible future performance degradation. This assessment procedure uses the CAST Application Intelligence Platform[3] (AIP) to automatically scan the implementation of these applications to review the architecture, design, and code against current industry best practices and known design flaws that may impact resilience. The AIP applies over 1000 engineering checks based on standards and measurements developed by the Software Engineering Institute (SEI), International Standards Organization (ISO), Consortium for IT Software Quality (CISQ), the Institute of Electrical and Electronics Engineers (IEEE) and the technology provider industry. The resulting analysis identifies specific flaws in the software and aggregates this information into metrics to objectively quantify the structural quality of the application.

The set of health factors used for the measurement are Robustness, Performance Efficiency, Security, Transferability, and Changeability (see Table 3.1). The Total Quality Index (TQI) combines all health factors into a single application quality score as an indication of the overall health of the application, this score is on a four-point scale, where 1.0 is a very risky application and a 4.0 is a very clean application. The final score is calculated through the detection of elements of a rule set of over 1000 software engineering industry norms, divided by each factor. Some of these norms are marked as being "critical", which reflects their high likelihood of turning into defects. Critical violations, thus, deserve specific attention and it is best to avoid introducing them in ongoing enhancement work. Every Health Factor is then calculated as a function of the rules violated, their weight/criticality, and the frequency of rule violations across all objects in the path of the transaction.

3.1.1 CAST Green IT Index

Due to the growing interest surrounding the Green-efficiency issue in software development, CAST offers a solution that is based on his Assessment Model: the *Green IT Index*. This Business Criterion aggregates a subset of Quality Rules from existing high-impact Efficiency-related Technical Criteria as well as from most-severe Robustness-related Technical Criteria to account for wasted resources when application functioning has been compromised and require a restart/recovery. The basic idea behind this Index is to detect those rules that

[2] Dr. R. Soley. OMG PhD MIT

[3] https://www.castsoftware.com/products/application-intelligence-platform

Table 3.1 Description of the health factors

Health Factor	Description Attributes	Example Business Benefits
Transferability	Allows new teams or members to quickly understand and work with an application	• Reduces inefficiency in transferring application work between teams • Reduces learning curves • Reduces lock-in to suppliers
Changeability	Makes an application easier and quicker to modify	• Improves business agility in responding to markets or customers • Reduces cost of ownership by reducing modification effort
Robustness	Affects the stability of the application and the likelihood of introducing defects when modifying it	• Improves availability of the business function or service • Reduces risk of loss due to operational malfunction • Reduces cost of application ownership by reducing rework
Performance	Affects the performance of an application	• Reduces risk of losing customers from poor Service or response • Improves productivity of those who use the application • Increases speed of making decisions and providing information • Improves ability to scale application to support business growth
Security	Affects an application's ability to prevent unauthorized intrusions	• Improves protection of competitive information-based assets • Reduces risk of loss in customer confidence or financial damages • Improves compliance with security-related standards and mandates

either lead to an unnecessary number of CPU cycles or that adds time to the computation. Unfortunately the definition of the index, even if motivated by the correct assumptions, is just an assemble of several quality rules belonging to different health factors. No empirical evaluation or motivation has been performed in order to justify their value from an energetic point of view.

3.2 Energy measurement: jRAPL

A central role in this work is represented by the measurement technique adopted for the profiling of energy consumption of the software under test. As shown in the previous chapter, literature offers different approaches to the problem, each of them giving pros and cons. Given the nature of the work, three principal requisites are selected for the approach selection process.

- *high frequency of sampling:* the test will consist in small amount of code so the execution time will be really small. Higher frequency can guarantee better results;
- *high precision:* for the same reason as the previous point, every sample will consist in a considerably small amount of power; higher precision is required to avoid excessively rounded or inaccurate results;
- *exportable results:* after the measurement step, is necessary to make some statistical analysis on the obtained data. This is only possible if these results can be saved and can be transferred in a virtual format, like a .csv or a log file.

The final decision was to use jRAPL library [32]. This library consists in a set of APIs that profiles energy for Java programs running on CPUs with RAPL (Running Average Power Limit) support. RAPL [11] is a set of low-level interfaces firstly introduced with Sandy Bridge microarchitecture with the ability to monitor, control, and get notifications of energy and power consumption data, providing accurate energy estimates reported in Joule at a very fine-grained level [19, 53]. RAPL is configured to read energy consumption information and store it in Machine-Specific Registers (MSRs), that can only be accessed by OS, such as the `msr` kernel module in Linux. jRAPL enables RAPL to also analyze power consumption of Java language.

jRAPL works for Intel microarchitectures of Skylake, Haswell, Sandy Bridge, Sandy Bridge_ep (Server) and Ivy Bridge. Each architecture has different RAPL sensor support, allowing it to read power consumption of different CPU-level components, like reported in Table 3.2; it is hence possible to monitor single components like CPU or DRAM or the sum of them, under the Package field. The user interface for jRAPL is extremely simple: it is just necessary to enclose the piece of code under energy analysis inside a pair of `statCheck()` method invocation, like the following example:

```
double beginning = EnergyCheck.statCheck();
// block of code whose energy information is under interest
double end = EnergyCheck.statCheck();
```

```
double total_energy = end - beginning;
```

microarchitecture	support
Sandy Bridge, Ivy Bridge	uncore GPU, CPU and package
Skylake, Haswell, Sandy Bridge_ep	DRAM, CPU and package

Table 3.2 RAPL support for different microarchitecture

Main advantages of using jRAPL library are:

- *Refined Energy Analysis:* it can also report breakdown among hardware components and among program components (such as methods and code blocks).
- It is specific for monitoring Java software.
- *Synchronization-Free Measurement:* the demarcation of measurement coincides with that of execution; no synchronization is needed.

3.3 Sources of detected patterns

The process of patterns identification looks into the definition of well known bugs. CAST and several Automatic Static Analysis (ASA) tools documentations reports some of the most common efficiency, robustness, and bad-practice violations.

3.3.1 CAST Documentation

CAST Documentation[4] offers a complete list of all the Quality Rules that are taken into account for the measurement of the Technical Quality indexes. These rules are divided into several categories depending on the Health Factor they influence: Changeability, Efficiency, Robustness, Security, Maintainability and Transferability.

3.3.2 Automatic Static Analysis Tools

For a more complete view on violation patterns, some of the main free automatic static analysis tools for Java applications are taken into account. Similarly to CAST Quality Rules, these tools report a list of violations that are considered bugs or potential weaknesses in applications under inspection. They perform a static analysis, i.e. without executing the code,

[4]http://doc.castsoftware.com/display/DOC82/Metrics+and+Quality+Rules+-+details

on application source code, with the intent of identifying a wide range of probable software anomalies. There are either commercial, open source, or developed by researchers tools, they principally vary in the number and in the type of concerns they can detect, as well as the programming languages they support.

FindBugs

FindBugs [5] is a free ASA tool developed at the University of Maryland. FindBugs uses static analysis to inspect Java bytecode for occurrences of potential faults, and maps these faults to the corresponding Java source code. It is written in Java, can be run with any virtual machine compatible with Sun's JDK 1.4 and can analyze programs written for any version of Java. Rules are classified in several categories: Bad Practices, Correctness, Malicius Code Vulnerability, Multithreaded correctness, Performance, Security and Dodgy code, and in additional 4 different ranks of gravity. According to its website, the ratio of false warnings is below 50% [6].

PMD

Similarly to FindBugs, PMD[7] is an open source static source code analyzer that finds common programming flaws and supports several languages: Java, JavaScript, Salesforce.com Apex and Visualforce, PLSQL, Apache Velocity, XML, XSL. PMD discovers suspicious or abnormal coding practices, which may imply serious bugs, by searching syntactic errors and stylistic conventions violations from source code. It also include CPD, a *copy-and-paste detector*, that permits to find duplicate code in multiple language source files. One of the biggest feature of this tool is the possibility to create custom bug patterns in order to scan code looking for particular, user-defined instructions. PMD divides its rules in 7 different categories: code style, documentation, error prone, multithreading, best practices, design, and performance.

[5] http://findbugs.sourceforge.net
[6] http://findbugs.sourceforge.net/factSheet.html
[7] https://pmd.github.io/pmd-6.0.0/index.html

Chapter 4

Green smells identification

This chapter illustrates the first phase of the work. The goal is to detect a set of Java code patterns that may represent significant energy hotspots. In the first part it is described the methodology that was adopted for the patterns selection. The second one lists al the identified patterns, reporting a brief description that characterizes the violation.

This work refers to *pattern* and *green smell* as two different concept. The first term denotes a portion of code that is suspected to represent an energy hotspot, while the second indicates a pattern that is proved to be energy inefficient. In contrast with the literature, that denotes energy hotspots as *energy code smells*, this work coins the term *green smells*. As Fowler described its code smells as "a surface indication that usually corresponds to a deeper problem in the software system" [14], green smells reflect a deeper problem concerning energy management. Each pattern is described as a rule that specifies the features of the bug, i.e. what *should not* be done for an energy efficient result.

4.1 Green smells identification

The main sources used for the identification of the initial set of suspicious patterns are CAST, Findbugs and PMD documentations. Among the high number of violations reported in these documentations, most attention is given to those rules related to efficiency, performance, robustness, bad practices and dodgy code rules.

4.1.1 Programming language choice

It was decided to only select patterns that concern the Java language. This choice was motivated by several reasons: CAST software can perform analysis on various programming languages, such as C, C++, C#, COBOL, SQL, Visual Basic and many others, including Java,

but even if Java does not represent the most widespread language, it appears to be among the most adopted for newly developed ones. Furthermore, when compared with other languages, it is one of the best option when it comes to energy consumption and performance time, and the best one among the non compiled ones [43].

4.1.2 Rules Selection Criteria

Before starting to look into violation specifications, some selection criteria are established in order to keep the quantity of code to be tested adequate and coherent to the goal. The criteria are intended for excluding all those rules that do not match the purposes of this work. The following are the adopted criteria:

- every violation must be resolvable, and the execution of code that contains the bug has to produce the same result as the execution that has the specified solution.
- violations that consist in either compile time or run time errors must be rejected. Faulty code is considered untestable, nevertheless it must be noted that code that causes frequent system crashes can quickly increase its carbon footprint. A better focus on the structural quality of software can improve applications efficiency as well as reduce all those defects that cause interruptions;
- rules specifying bugs that, even if not generating any error, do not have any solution must not be taken into account.
- “coding style”rules, i.e. all those best practices or naming convention that does not interfere with the effective computation, are rejected. This set of bugs, when does not interfere with syntactic rules, is logically not intended to affect energy management in any way;
- rules without a fixed solution must not be taken into account. Solutions that are too general, or vary according to specific situations, are not good for a valid test case;
- the selection of the rules also follows some sort of intuition. Rules that intuitively do not interfere with energy efficiency at all are discarded from the selection.

The application of these criteria was as strict as possible, but in some occasions a bit of flexibility was applied. For example, pattern specifically intended for other programming languages were in some occasion readapted to conform to Java syntax; when no general remedies were provided, a customized solution was studied.

4.2 Identified patterns

After an accurate analysis of all the documentations taken into account, an initial list of patterns to be tested is created. The detected patterns has been divided into different categories, for grouping all those violations belonging to the same semantical area, as reported in Table 4.1. These divisions help to better evaluate the results of the testing phase, and to confront the energy spending differences between similar patterns.

Table 4.1 Identified patterns

section	method identifier
Expensive Calls in Loops	callPropertiesThatCloneValuesInLoop; indirectExceptionHandlingInsideLoops; indirectStringConcatenationInLoops; instantiationInsideLoop; stringConcatenationInLoop; terminationExpressionInLoop;
DB calls	closeDBResourcesASAP; usingLONGDatatype; nullableColumn; selectAll; implicitConversionInWHERE; preferUNIONALL; usingOldStyleJoin; mixAnsiWithOracleSyntax; existsIndipendentClause; withoutPrimaryKey; useDedicatedStoredProcedure; sqlQueriesInsideLoop; tooManyIndexes; firstIndexInWhere; redundantIndexes;
Expensive Object Instantiation	dynamicInstantiation; instantiatingBoolean; largeNumberOfStringConcat; stringInitializationWithObject; declareStaticMethod; boxingImmediatelyUnboxed; numberCtor; randomUsedOnlyOnce; needlessInstantiation; boxedPrimitiveToString; newObjectForGetClass;
Garbage Collector calls	callSuperFinalizeInFinally; callingFinalize;
Poor Programming Choices	arrayCopy; collapsibleIfStatement; deadLocalStoreReturn; formatStringNewLine; localDoubleAssignment; mutualExclusionOR; nonShortCircuit; repeatedConditionals; selfAssignment; staticFieldCollection; switchRedundantAssignment; useArraysAsList; uselessControlFlow; uselessSubstring; usingHashtable; usingRemoveAllToClearCollection; usingStringEmptyForStringTest; usingVector; variableDeclaredAsObject;

Every pattern is here explained in plain language, briefly reporting its features and its corresponding remediation. A better and more accurate description will be proposed in next

chapters, regarding only the patterns that, after the test results analysis, effectively correspond to energy hotspots and are thus namely defined as green smells.

4.2.1 Expensive calls in loops patterns

- **callPropertiesThatCloneValuesInLoop**[1]**:** accessing through a loop to a property with a method that returns cloned objects is a computing intensive task. Assigning the property to a local variable outside the iteration can reduce the overhead.
- **indirectExceptionHandlingInsideLoops**[2]**:** indirect calls to loops that contains try-catch blocks can be really computing intensive. On the contrary, it is less expensive to put the loop inside the try-catch block.
- **indirectStringConcatenationInLoops**[3]**:** functions that perform String concatenation inside loops are among the highest inefficient operations, for reasons better explained later. Performance results in this kind of situations are also worsened by indirect calls. A solution for reducing the overhead may be to create `StringBuffer` object before entering the loop, and then append intermediate results to it.
- **instantiationInsideLoop**[4]**:** the instantiation of Object in Java is quite an expensive operation, but this downside is easily overcame by the benefits that they bring to object-oriented programming. Nevertheless, performing creation of several temporary Objects inside loops reflects poor programming choices, and leads to serious overhead.
- **stringConcatenationInLoop**[5]**:** similarly to the previously explained patterns, large number of string concatenations in sequence may severely affect computation efficiency and resource management. A better solution is to use StringBuffer Object to decrease the overhead.
- **terminationExpressionInLoop**[6]**:** termination conditions in a loop that use a method call can induce a significant overhead. Not only the call of method comes with a

[1]http://doc.castsoftware.com/display/DOC82/Avoid+calling+properties+that+clone+values+in+loops
[2]http://doc.castsoftware.com/display/DOC82/Avoid+indirect+exception+handling+inside+loops
[3]http://doc.castsoftware.com/display/DOC82/Avoid+indirect+String+concatenation+inside+loops
[4]http://doc.castsoftware.com/pages/viewpage.action?pageId=161550518
[5]http://doc.castsoftware.com/pages/viewpage.action?pageId=164499727
[6]http://doc.castsoftware.com/display/DOC82/Avoid+method+invocation+in+a+loop+termination+expression

important cost but depending on the complexity and the performance of the called method, the impact on performance can be huge. Indeed, the method can be called several hundred to several thousand times, depending on the size of the loop. Using a local variable to store the condition and terminating the loop accessing it improves computation efficiency.

4.2.2 DB calls patterns

Other patterns that are not entirely related to Java language were also taken into account. This second section of possible green smells concerns database interaction and the relative retrieved data manipulation. Unfortunately it is not possible to also measure the energy consumed by the database-side of these procedures, because of the measurement tools limitations. Despite this fact, these pattern will be analyzed in order to detect improvements on the Java-side of the connection.

- **closeDBResourcesASAP**[7]**:** incorrect resource management is a common source of failures in production applications, with the usual pitfalls being database connections and file descriptors remaining open after an exception has occurred somewhere else in the code. This leads to application servers being frequently restarted when resource exhaustion occurs, because operating systems and server applications generally have an upper-bound limit for resources. Better to always check to close all the open connection with a `finally` block.

- **existsIndipendentClause**[8]**:** SQL objects using EXISTS which are independent, i.e. not referring to the parent object, are a potential performance and disk space issue. Always check for this type of violations.

- **firstIndexInWhere**[9]**:** whenever a query with a WHERE clause does not use the first column of composite index (multiple column base index), the database engine will not use the composite index to retrieve the data, potentially leading to very poor performance. Depending on the context, the user should evaluate the fact that the composite index is leading to poor performance. If the query does not use other indexes, the fact that the query does not use the first column of a composite index is most probably an error that needs to be fixed.

[7] http://doc.castsoftware.com/display/DOC82/Close+database+resources+ASAP
[8] http://doc.castsoftware.com/display/DOC82/Avoid+exists+independent+clauses
[9] http://doc.castsoftware.com/display/DOC82/Avoid+SQL+queries+not+using+the+first+column+of+a+composite+index+in+the+WHERE+clause

- **implicitConversionInWHERE**[10]**:** a query using implicit type conversion usually introduces performance issues into an application as well as classical data conversion issues. Typical example is DATE conversion from a String format.

- **mixAnsiWithOracleSyntax**[11]**:** ANSI syntax allows a clear separation between JOIN clause and WHERE clause restrictions. The ANSI notation makes the relations between the tables explicit, and saves you from coding equality tests for JOIN conditions in the WHERE clause. Thus, the mix of both JOIN notations makes the readability and maintainability of the code more complex. Moreover ANSI JOIN allows having optimization hints, so it is better to transform the queries including the two different notations with only the use of the ANSI syntax.

- **nullableColumn**[12]**:** columns marked as "NULLABLE" should be last in the table column order. Placing columns that frequently contain NULLs last in the table column order, minimizes the average row length and optimizes the table data density, moreover it also minimizes the number of column length bytes that need to be navigated to access the non-NULL column values. This factors benefits performances.

- **preferUNIONALL**[13]**:** in SQL the UNION operator combines the results of two SQL queries into a single table of all matching rows. The two queries must have matching fields and data types in order to join them. Any duplicate records are automatically removed unless UNION ALL is used. According to the data set in use, most of the time it is unnecessary to remove duplicates as there are none or only a few. Since doing the search for duplicate rows can be very costly, unless strictly required is preferable to use UNION ALL.

- **redundantIndexes**[14]**:** it is useless to have several indexes on the same column(s) or to have an index on columns already covered by a composite index. This will not improve read performance and will decrease update/insert/delete performances because of index maintenance. Always consider to drop the index that are not strictly necessary, or that do not introduce any performance gain.

[10]http://doc.castsoftware.com/display/DOC82/Avoid+SQL+queries+with+implicit+conversions+in+the+WHERE+clause
[11]http://doc.castsoftware.com/display/DOC82/Do+not+mix+Ansi+joins+syntax++with+Oracle+proprietary+joins+syntax+in+the+same+query
[12]http://doc.castsoftware.com/pages/viewpage.action?pageId=161550722
[13]http://doc.castsoftware.com/display/DOC82/Prefer+UNION+ALL+to+UNION
[14]http://doc.castsoftware.com/display/DOC82/Avoid+redundant+indexes

- **selectAll**[15]**:** a query that retrieves all columns of a table with a SELECT * can potentially be the source of important changeability problems. This can lead to important data inconsistencies and thus stability issues. Performance problems may also arise when the execution of the query returns a large result sets (many row with all the columns may then become a huge amount of data to transport over the network). Thus optimizer module can not provide a correct execution.

- **sqlQueriesInsideLoop**[16]**:** having an SQL query inside a loop is usually the source of performance and scalability problems especially if the number of iterations become very high (for example if it is dependent on the data returned from the database). This iterative pattern has proved to be very dangerous for application performance and scalability. Database servers handle in a much better set-oriented pattern rather than pure iterative ones.

- **tooManyIndexes**[17]**:** every index increases the time it takes to perform INSERTS, UPDATES and DELETES, so the number of indexes should not be too high. The amount on each single table should never be more then 4/5 indexes. On the other hand, for read-only table the number of indexes is not as important because does not affect performance and can so be larger.

- **useDedicatedStoredProcedure**[18]**:** software that does not leverage database capabilities to efficiently run data processing (such as stored procedures and functions) requires excessive computational resources. Dedicated stored procedures are preferred when multiple data accesses are needed.

- **usingLONGDatatype**[19]**:** Long and Long Raw datatypes should not be used for table columns. According to Oracle documentation, LOB datatype is the preferred option for several reasons: LONG can only store 2GB whereas LOB can have 4GB; in PL/SQL only 32760 bytes can be handled on LONG (on possibly 2GB); access to LONG are sequential whereas access to LOB can be direct (thus having better performances).

- **usingOldStyleJoin**[20]**:** as already illustrated before, it is preferrable to use the ANSI-style Join operator rather then the old style Oracle one. ANSI syntax allows a clear

[15] http://doc.castsoftware.com/pages/viewpage.action?pageId=161550369
[16] http://doc.castsoftware.com/display/DOC82/Avoid+using+SQL+queries+inside+a+loop
[17] http://doc.castsoftware.com/display/DOC82/Avoid+too+many+Indexes+on+one+Table
[18] http://doc.castsoftware.com/pages/viewpage.action?pageId=164500112
[19] http://doc.castsoftware.com/pages/viewpage.action?pageId=164499868
[20] http://doc.castsoftware.com/pages/viewpage.action?pageId=164499700

separation between JOINs clauses and the WHEREs clauses restrictions. The ANSI notation makes the relations between the tables explicit, saves you from coding equality tests for JOIN conditions in the WHERE clause, and allows having optimization hints.

- **withoutPrimaryKey**[21]**:** in relational database design, a candidate key is just a unique identifier. A primary key is a candidate key that has been singled out to uniquely identify each row in a table. A unique key or primary key comprises a single column or set of columns. No two distinct rows in a table can have the same value (or combination of values) in those columns, so they have great performance impact when used as indexes.

4.2.3 Expensive object instantiation patterns

- **boxedPrimitiveToString**[22]**:** it is inefficient to allocate a boxed primitive Object only for calling the `toString()` method. A smarter solution is to directly call the static form of `toString()`.

- **boxingImmediatelyUnboxed**[23]**:** an immediate conversion of a primitive boxed value makes the first instantiation useless, just perform a direct primitive coercion instead. e.g. `new Double(d).intValue()`.

- **declareStaticMethod**[24]**:** all methods that do not use instance fields and/or methods should be declared as `static`, except if they have been extended in a subclass or inherit from their parents. This procedure makes the instantiation unnecessary, avoiding memory and resource allocation.

- **dynamicInstantiation**[25]**:** dynamic instantiations (i.e. through calls of `newInstance()`, `getMethod()`, `getField()` or `invoke()` on Object of Class, Method or Constructor) should be used only when strictly necessary, because it is slower than a regular Class invocation or Method call.

[21] http://doc.castsoftware.com/display/DOC82/Avoid+Tables+without+Primary+Key
[22] http://findbugs.sourceforge.net/bugDescriptions.html#DM_BOXED_PRIMITIVE_TOSTRING
[23] http://findbugs.sourceforge.net/bugDescriptions.html#BX_BOXING_IMMEDIATELY_UNBOXED_TO_PERFORM_COERCION
[24] http://doc.castsoftware.com/pages/viewpage.action?pageId=164499970
[25] http://doc.castsoftware.com/display/DOC82/Avoid+using+Dynamic+instantiation

- **instantiatingBoolean**[26]**:** the instantiation of Boolean Objects is an avoidable operation that consumes memory and CPU but does not bring value. Using `Boolean.TRUE` or `Boolean.FALSE` has the same logical effect and does not waste resources.

- **largeNumberOfStringConcat**[27]**:** all the String Classes should not call the + method for concatenation in sequence. String concatenation resolved at runtime is much slower than using StringBuffer `append()` method.

- **needlessInstantiation**[28]**:** Object allocation of Classes that only supplies static methods are useless and represent a waste of resources and an avoidable constructor invocation. A better and correct remedy is to simply access the static methods directly using the Class name as a qualifier.

- **newObjectForGetClass**[29]**:** allocating a new Object only for invocating its `getClass` method for retrieving the Class Object represents a waste of resource. Better to simply access the `.class` property of that Class.

- **numberCtor**[30]**:** using `new` on Number Classes instances (`Integer`, `Long`, `Double`, `Float`, `Short`, `Byte`) like `new Integer(int)`, results in a new Object, whereas using `Integer.valueOf(int)` allows caching of values to be done by the compiler, class library or JVM. This second option avoids Object allocation and improves code efficiency.

- **randomUsedOnlyOnce**[31]**:** methods that generate casual values should not create new `Random` Object for each time they are invoked. This approach produces mediocre quality random numbers and is inefficient. On the contrary, creating the `Random` Object only once, saving it, and invoking the appropriate method on it at each time a new number is required is more efficient and produce better results.

- **stringInitializationWithObject**[32]**:** initializations of String Objects using the String Class constructor is less efficient and slower then to directly use the literal value.

[26] http://doc.castsoftware.com/display/DOC82/Avoid+instantiating+Boolean
[27] http://doc.castsoftware.com/pages/viewpage.action?pageId=161550544
[28] http://findbugs.sourceforge.net/bugDescriptions.html#ISC_INSTANTIATE_STATIC_CLASS
[29] http://findbugs.sourceforge.net/bugDescriptions.html#DM_NEW_FOR_GETCLASS
[30] http://findbugs.sourceforge.net/bugDescriptions.html#DM_NUMBER_CTOR
[31] http://findbugs.sourceforge.net/bugDescriptions.html#DMI_RANDOM_USED_ONLY_ONCE
[32] http://doc.castsoftware.com/pages/viewpage.action?pageId=164499728

4.2.4 Garbage collector calls patterns

- **callSuperFinalizeInFinally**[33]**:** all `finalize` methods should call `super.finalize` to ensure that any superclass `finalize` methods are invoked. Its calls should also be made from a `finally` block to ensure that it is called regardless of whether the call to the cleanup method generates an exception.

- **callingFinalize**[34]**:** `finalize()` methods are supposed to be invoked at most one time by the garbage collector on Objects that are no longer referenced, in order to release unused memory resources. Explicit invocations of finalizers ingore the current state of the objet and do not modify it from unfinalized or finalizable to finalized. A better programming choiche is to remove the explicit call and let the Garbage Collector to handle the memory resources allocations.

4.2.5 Poor Programming Choices patterns

- **arrayCopy**[35]**:** when a copy of an array into another is required, a solution that loops through all the element of the first one and inserts them one by one into the other is highly inefficient. It is better to use dedicated methods like `System.arrayCopy()`.

- **collapsibleIfStatement**[36]**:** if the body of conditional branches permits it, two consecutive `if` statements can be consolidated by separating their conditions with a boolean short-circuit operator.

- **deadLocalStoreReturn**[37]**:** an assignment in a return statement has an effect, and it can make previous operations useless.

- **formatStringNewLine**[38]**:** when using format strings, it is preferable to use `%n`, which will produce the platform-specific line separator, instead using a newline character `\n`.

- **localDoubleAssignment**[39]**:** assigning the same value to a variable twice is useless, and may indicate a logic error or typo.

[33] http://doc.castsoftware.com/pages/viewpage.action?pageId=161550827
[34] http://doc.castsoftware.com/pages/viewpage.action?pageId=161550402
[35] http://www.precisejava.com/javaperf/j2se/Loops.htm
[36] https://pmd.github.io/pmd-6.0.0/pmd_rules_java_design.html#collapsibleifstatements
[37] http://findbugs.sourceforge.net/bugDescriptions.html#DLS_DEAD_LOCAL_STORE_IN_RETURN
[38] http://findbugs.sourceforge.net/bugDescriptions.html#VA_FORMAT_STRING_USES_NEWLINE
[39] http://findbugs.sourceforge.net/bugDescriptions.html#SA_LOCAL_DOUBLE_ASSIGNMENT

- **mutualExclusionOR**[40]**:** Mutual exclusion over the OR operator that always evaluates to true performs useless operations.

- **nonShortCircuit**[41]**:** using non-short-circuit logical operators (&, |) can be a programming choice, but when their usage is not justified leads to a non-optimized boolean evaluation. Non-short-circuit logic causes both sides of the expression to be evaluated even when the result could be inferred from only knowing the left-hand side. When it is possible, is better to use short-circuit ones. Understandably, this rule may results in a large number or False Positive, that is all the occurrence of non-short-circuit operator specifically intended for their purpose. For this reason, this rule is only intended to assess the difference in energy consumption between the usage of the two different applications of the operators.

- **repeatedConditionals**[42]**:** a conditional test that is performed multiple times does not affect the computation, so it is needless computation.

- **selfAssignment**[43]**:** an operation that assigns a variable to itself has no effect whatsoever.

- **staticFieldCollection**[44]**:** static collection Classes, such as HashMap and Vector, can cause memory leaks. Static collections are likely to cause memory leaks because static variables remain in memory as long as the application runs, regardless of its object creation and destruction. So because their life cycle equals the application duration, the objects that they reference will be kept in memory until the application ends, causing an overhead. Possible solutions are either to check that Objects added in the collection are removed when required, or to use collections with weak references (like the WeakHashMap Class). This will leverage the garbage collector's ability to determine reachability of referenced Objects for you.

- **switchRedundantAssignment**[45]**:** redundant assignments in a switch statement perform needless branches. Just merge all the equivalent cases in a single switch branch.

[40] http://cppcheck.sourceforge.net/
[41] http://findbugs.sourceforge.net/bugDescriptions.html#NS_NON_SHORT_CIRCUIT
[42] http://findbugs.sourceforge.net/bugDescriptions.html#RpC_REPEATED_CONDITIONAL_TEST
[43] http://cppcheck.sourceforge.net/
[44] http://doc.castsoftware.com/display/DOC82/Avoid+static+Field+of+type+collection
[45] http://cppcheck.sourceforge.net/

- **useArraysAsList**[46]**:** when it comes to populate a List with all the elements of an Array, copying all these items through a loop over the Array is a very poorly efficient operation. The `java.util.Arrays` Class provides a `asList` method that fastly fulfills the purpose.

- **uselessControlFlow**[47]**:** useless control flow statements, where control flow continues onto the same place regardless of whether or not the branch is taken, are useless. This is usually caused by having an empty statement block for an `if` statement.

- **uselessSubstring**[48]**:** invocations of `substring(0)` on a String have no effects, since they return the original String value. Direct uses of the original Object have the same effects, without any invokation of the Method.

- **usingHashtable**[49]**:** since HashTables are synchronized objects, they are much slower then the various List and Map implementations. When their usage is not strictly needed nor intended, it is a smarter choice to use other Collections instead.

- **usingRemoveAllToClearCollection**[50]**:** removing all the elements from a Collection for clearing it can be a wasteful operation. Invoking `c.removeAll(c)` on a Collection c is a less clear, susceptible to error from typos and less efficient choice. Instead, `c.clear()` results in a smarter and safer performance.

- **usingStringEmptyForStringTest**[51]**:** `String.Empty` should not be used for empty String tests. Comparing String length with 0 has twice better time performances; otherwise using `String.IsNullOrEmpty` Method can also improve robustness of the code.

- **usingVector**[52]**:** using thee Vector collection can lead to performance problems. Since Vectors are all synchronized –yet synchronization is usually not needed–, they are much slower than the various List and Map implementations, so when synchronization is not strictly needed, it is better to prefer List or Map adaptations.

[46] https://pmd.github.io/pmd-6.0.0/pmd_rules_java_performance.html#usearraysaslist
[47] http://findbugs.sourceforge.net/bugDescriptions.html#UCF_USELESS_CONTROL_FLOW
[48] http://findbugs.sourceforge.net/bugDescriptions.html#DMI_USELESS_SUBSTRING
[49] doc.castsoftware.com/display/DOC82/Avoid+using+Hashtable
[50] http://findbugs.sourceforge.net/bugDescriptions.html#DMI_USING_REMOVEALL_TO_CLEAR_COLLECTION
[51] http://doc.castsoftware.com/display/DOC82/Avoid+using+String.Empty+for+empty+string+tests
[52] http://doc.castsoftware.com/display/DOC82/Avoid+using+Vector

- **variableDeclaredAsObject**[53]**:** instantiations directly from general Class Object when not explicitly needed can waste several bytes of memory.

4.2.6 Considerations on the identified patterns

A considerable number of patterns concerns inefficient Object instantiations. One of the fundamental OO performance management principles is to avoid excessive object creation. This does not mean to abstain from using the benefits of object-oriented programming by not creating any objects, but it rather suggests to be wary of object creation risks when executing performance-critical code –like for example inside of loops–. The creation of any Object involves memory allocation for all its fields, all its superclasses fields and all its subclasses fields; the superclasses ones are initialized and the Class constructor is then invoked. From the amount of operations involved, it is evident that Object creation is a procedure expensive enough to reconsider its usage in temporary or intermediate objects. If it is not necessarily required, it should always be considered to use an equivalent static.

In relation to the instantiation problem there is also the issue of String concatenation. This represents a risky operation because of the internal activities it comprehend: during each concatenation between two Strings `a` and `b`, a new StringBuilder Object `c` is created. Firstrly empty, it is then appended with the values of `a` and `b`, and eventually re-casted to a String for the return. This –only apparently– trivial operation actually consists in a series of non-negligible ones, and if used serially it results in the creation of a large number of temporary Objects.

Additional attention has to be put when sensible operations are executed inside loops. Operation that may seems “harmless” became performance-affective when executed in large number inside `for` or `while` loops. For computing intensive operations like the already mentioned Object instantiation and String concatenation, embodying them in loops notably increases the impact they have on the performance. It is interesting to analize how the energy spending is affected and whether it rapidly increments.

Some patterns are related to the choice in using certain methods rather then similar others, according to specific situation. Using the method `c.clear` rather then preferring `c.removeAll(c)` when clearing a Collection `c` has no difference at all in the final results. But for how the methods are implemented, their performance is different. Equivalently, some similar methods are tested for telling which solution is better for an energy saving approach.

Garbage Collector is a controversial subject in Java. Directories say that the garbage collector should not be explicitly used in the code, since it is an automated process scheduled

[53] doc.castsoftware.com/display/DOC82/Avoid+Variables+declared+as+Variants

by the Java Runtime Environment. Its execution is non-deterministic, so there is no way to predict when garbage collection will occur at run time. It is although possible to include hints in the code to run the garbage collector with the System.gc() or Runtime.gc() methods, but they provide no guarantee that the Garbage Collector will actually run. The purpose of the patterns here defined is then to determine whether these calls have some effect on the actual invocation of the deletion process and thus how much they affect the energy spending.

Pattern that tests database connection handling issue with resource leak or data manipulation. Both situation are factors that can lead to inefficiencies related to also the Java side of the connection. On the other side, for example, executing all the energy-intensive operation on the database side of the communication rather than the Java side can lead to several benefits. Firstly, SQL and the other query languages have better capability in managing data and structures than Java. Furthermore, data results optimizations on that side leads to transfer smaller quantity of data. All these factor have a positive impact on the total energy consumption. For better results, each database-related pattern has been implemented for three different management systems: MySQL 14.14, PostgreSQL 9.5.12 and Oracle 11g Express Edition 11.2.0.2.0. In this way results can be more generalized and free of any platform-specific influence.

Chapter 5

Validation of the identified green smells

This chapter first explains the experimental setup and then gives an analysis on the final outcome of the performed tests. There are two main research question that are addressed in this phase of the work:

- **RQ1:** which of the patterns under test represent an energy hotspot (i.e. which are green smells)?
- **RQ2:** which are the main variables among execution time, resource usage and bytecode instruction, that are responsible of the increase in energy consume?

Both the research questions are discussed in this chapter, with a final analysis on the obtained results.

5.1 Preliminary setup

Each pattern is tested for establishing whether it represents a green smell or not. For each identified pattern, a couple of Java methods is created: one that implements the violation, the other that implements the corresponding resolution. The goal of the experiment is to measure the amount of power that each element of the pair spends, and then evaluate if the resolution of the violation affects the energy consumption. If the energy that is consumed results improved with the resolution of the violation, than the pattern is considered a green smell.

Every pattern is implemented in a Java method, using the following signature:

public static void patternName_hasSmell_source()

where *hasSmell* can assume two possible values: *With*, if the method implements the bugged version, *Without* if it represents the resolved counterpart. *source* also assumes different values according to the the documentation source it was taken from. For example, the signatures of the pair of methods that tested the *terminationExpressionInLoop* pattern, which comes from CAST documentation, have the following structure:

```
public static void terminationExpressionInLoop_With_CAST(){
    int[] arr = new int[500];
    for (int i=0; i<arr.length;  i++) {
        arr[i] = i;
    }
}

public static void terminationExpressionInLoop_Without_CAST(){
    int[] arr = new int[500];
    for (int i=arr.length-1; i>=0; i--) {
        arr[i] = i;
    }
}
```

In order to minimize the interferences that can occur among different test units, every method is implemented as independent to the others and no Class or structure is shared between different patterns.

Collecting data for each single function represents a challenging task because of the dimension of the patterns under test. Most of them consists in only a couple of lines of code, so their execution is too fast to provide a reliable and significant amount of data. For this reason, every pattern is repeated for 1 million times, collecting enough consistence for making a significant sample. In order to reach statistical significance too, 50 samples are collected for every function. At the end, for each pattern are collected 100 sample: 50 for the function with the inefficient pattern and 50 for the resolved one. For reducing the noise and the interferences between the measurements, the tests are launched one at time, independently to the others, minimizing the processes running in background on the operating system. The experiment tests are executed on a laptop with the following technical specifications:

- HP Pavilion x360 - 13-a105nl with 4×Intel Core i3-4030U @1.90GHz, 8GB DDR3 RAM, 500GB SSHD with Ubuntu 16.04 LTS 64-bit

A pair of hypothesis are defined for each test, a null and an alternative one. The hypothesis are then tested using the Mann-Whitney (or Wilcoxon-Mann-Whitney) test [33, 20]. Given two random variables x and y with cumulative distribution functions f and g, this test decides whether $f = g$ against the alternative hypothesis that x is stochastically smaller than y. x is said to be stochastically smaller than y if $f(a) > g(a)$ for every a. In other words the mean of the test is to determine whether two independent samples belongs to two populations that have the same distributions or not. Mann-Whitney test is often presented as an alternative to a t test when the data are not normally distributed. Whereas a t test is a test of population means, the Mann-Whitney test is commonly regarded as a test of population medians. This is not strictly true, because other factors like differences in spread may be as well important. Results with a p-value that are smaller then a given α indicate that one of the two populations under analysis has a bigger distribution than the other, without specifying which one.

5.2 Validation of green smells

5.2.1 Experiment setup

A pair of hypothesis, a null one and an alternative one, is formulated for each pattern under test. The results help the identification of the patterns that are green smells. The hypothesis are the following:

$H1_0$: E_{with} and $E_{without}$ are similar

$H1_a$: E_{with} and $E_{without}$ are independent

where E is the energy consumption of the pattern under test, with or without the potential smell. The null hypothesis is rejected in favor of the alternative one only if one of the pattern consumes less or more energy then its relative counterpart. The hypothesis is tested with a given $\alpha = 0.05$. If this condition is met, the impact that the resolution of the bug has on energy consumption decides whether a green smell is found or not. Positive impacts (i.e. smaller than zero) reflect a reduction in the overall consumption, detecting a green smell; on the other hand, negative ones (i.e. greater than zero) indicate an increase. As a precaution against possible noise during energy measurements, a threshold was imposed in order to considers relevant only the results with an impact greater than 5% on the overall value.

The set of patterns related to database interactions is tested separately from the others because its validation requires additional settings. Since this patterns are implemented in multiple versions, according to three specific management systems (MySQL, PostgreSQL

and Oracle), an additional condition is states that patterns are considered green smells only if they experience an energy improvement on all the tested technologies. This condition is necessary in order to minimize platform-specific optimization and to make the green smells as general as possible.

5.2.2 Results

The patterns are divided into the categories already defined in the previous chapter, the Table reports the name of the pattern, the means of the power consumption for the function with the violation (*mean w*), and without the violation (*mean w/o*) in Joule, the difference in percentage (*impact*) of power consumption, and the p-value of the Mann-Whitney test. The rows of the patterns that satisfy both the requirement (impact < -5 and p-value < 0.05) are marked in bold, indicating that a green smell is found. In results is also reported an additional column named *rank*. This value divides only the validated green smells into different values, according to the gravity of the impact they have on energy consumption. If the *impact* they have is heavier, then the rank is reported higher, otherwise if they have a minimal impact rank value is smaller.

Experiment results are reported in Tables 5.1, from the total 38 patterns, 19 (marked in bold) are proven to be green smells. The largest number of the smells belongs to two categories of patterns: *Expensive Object Instantiation* and *Expensive Calls in Loops*.

Instantiation of objects that are no strictly necessary is proved to bring a large waste in energy resources. Object allocation involves some collateral effects that have a non negligible impact on the computation, such as the assignment of the required memory, the invocation of Class constructors and so on. It is straightforward that avoiding an instantiation prevents all these expensive instructions to be executed. All the *Expensive Object Instantiation* patterns result to be green smells, apart from *largeNumberOfStringConcat*.This outcome contrasts with the other String-concatenation patterns, that contrariwise reflect a good improvement rate of the `StringBuffer.append()` solution against the + operator. The motivation resides in the compiler behaviour: operations using + operator are resolved at compile time, while `append()` method at run time. Run time resolution takes place when the value of the String is not known in advance whereas compile time resolution happens when the value of the String is known in advance. In this situation, + operator results in more efficient behaviour, but when they are compiled at run time, like when both in a loop as in *stringConcatenationInLoop*, StringBuffer works more efficiently. The + operator at run time results in bad performances because it actually involves the creation of temporary Objects that affect the final execution. Appending a String `b` to a `(a += b)` is indeed equivalent to `x = new StringBuilder()`, `x.append(a)`, `x.append(b)`, `x.toString();`.

Table 5.1 Energy results

method name	mean w	mean w/o	impact	p-val	rank
callPropertiesThatCloneValuesInLoop	**15.174**	**4.121**	**-114.568**	**6.86e-18**	**9**
indirectExceptionHandlingInsideLoops	**31.309**	**6.529**	**-130.980**	**6.86e-18**	**9**
indirectStringConcatenationInLoops	**5.907**	**3.295**	**-56.775**	**6.86e-18**	**9**
instantiationInsideLoop	**9.705**	**7.734**	**-22.608**	**6.86e-18**	**4**
stringConcatenationInLoop	**2.321**	**2.065**	**-11.693**	**6.86e-18**	**7**
terminationExpressionInLoop	3.965	3.913	-1.321	2.29e-04	
boxedPrimitiveToString	**4.981**	**3.497**	**-35.003**	**6.86e-18**	**7**
boxingImmediatelyUnboxed	**0.350**	**0.284**	**-20.863**	**4.54e-13**	**4**
declareStaticMethod	**0.370**	**0.327**	**-12.357**	**6.86e-18**	**4**
dynamicInstantiation	**1.139**	**0.343**	**-107.529**	**6.86e-18**	**9**
instantiatingBoolean	**0.356**	**0.295**	**-18.723**	**6.34e-15**	**4**
largeNumberOfStringConcat	1.170	1.819	43.412	6.86e-18	
needlessInstantiation	**0.300**	**0.284**	**-5.442**	**1.67e-02**	**2**
newObjectForGetClass	**0.315**	**0.254**	**-21.349**	**1.59e-14**	**2**
numberCtor	**0.352**	**0.310**	**-12.733**	**1.01e-07**	**4**
randomUsedOnlyOnce	**46.081**	**11.575**	**-119.694**	**6.86e-18**	**9**
stringInitializationWithObject	**0.381**	**0.297**	**-24.882**	**1.05e-15**	**4**
callSuperFinalizeInFinally	11.385	11.427	0.371	6.86e-18	
callingFinalize	16.679	16.769	0.535	6.86e-18	
arrayCopy	6.552	6.350	-3.123	8.71e-14	
collapsibleIfStatement	0.295	0.282	-4.669	1.31e-01	
deadLocalStoreReturn	0.274	0.267	-2.696	4.76e-01	
formatStringNewLine	101.758	102.519	0.745	1.95e-01	
localDoubleAssignment	0.309	0.319	3.300	5.84e-02	
mutualExclusionOR	0.256	0.262	2.351	3.45e-01	
nonShortCircuit	**0.353**	**0.269**	**-26.964**	**8.73e-06**	**4**
repeatedConditionals	0.258	0.261	0.892	6.62e-01	
selfAssignment	0.269	0.260	-3.065	2.02e-01	
staticFieldCollection	1.283	1.801	33.584	6.86e-18	
switchRedundantAssignment	0.265	0.268	0.800	8.47e-01	
useArraysAsList	**6.906**	**0.859**	**-155.739**	**6.86e-18**	**9**
uselessControlFlow	0.268	0.262	-2.367	2.82e-01	
uselessSubstring	0.315	0.300	-4.746	1.13e-01	
usingHashtable	**18.952**	**10.651**	**-56.087**	**6.86e-18**	**9**
usingRemoveAllToClearCollection	**6.622**	**4.580**	**-36.459**	**6.86e-18**	**7**
usingStringEmptyForStringTest	0.315	0.309	-1.863	3.33e-01	
usingVector	5.177	5.287	2.105	6.38e-05	
variableDeclaredAsObject	2.916	2.902	-0.512	2.44e-01	

Table 5.2 Energy DB results

method name	mean w	mean w/o	impact	p-value	rank
closeDBResourcesASAP_MYSQL	3.81	3.74	-1.8177	6.77e-02	
existsIndipendentClause_MYSQL	**6.25**	**3.58**	**-54.1686**	**6.86e-18**	**9**
firstIndexInWhere_MYSQL	14.19	14.14	-0.4009	1.08e-01	
implicitConversionInWHERE_MYSQL	3.81	3.79	-0.5157	1.97e-01	
mixAnsiWithOracleSyntax_MYSQL	4.34	4.32	-0.4951	1.32e-01	
nullableColumn_MYSQL	3.69	3.70	0.2052	5.84e-01	
preferUNIONALL_MYSQL	3.67	3.65	-0.6397	1.25e-01	
redundantIndexes_MYSQL	13.82	13.85	0.1856	5.13e-01	
selectAll_MYSQL	3.83	81.26	182.0021	6.25e-01	
sqlQueriesInsideLoop_MYSQL	**17.08**	**3.99**	**-124.3019**	**6.86e-18**	**9**
tooManyIndexes_MYSQL	25.45	24.58	-3.4868	4.00e-01	
useDedicatedStoredProcedure_MYSQL	5.99	6.07	1.2112	3.23e-10	
usingLONGDatatype_MYSQL	4.47	4.47	-0.1997	8.04e-01	
usingOldStyleJoin_MYSQL	4.08	4.08	0.0838	7.41e-01	
withoutPrimaryKey_MYSQL	3.57	75.99	182.0351	4.18e-01	
closeDBResourcesASAP_POSTGRE	2.06	2.04	-1.1264	5.53e-01	
existsIndipendentClause_POSTGRE	**4.47**	**1.98**	**-77.3015**	**6.86e-18**	**9**
firstIndexInWhere_POSTGRE	12.24	12.18	-0.4994	2.25e-01	
implicitConversionInWHERE_POSTGRE	2.08	2.15	3.2811	2.95e-05	
mixAnsiWithOracleSyntax_POSTGRE	1.96	1.96	-0.0921	8.04e-01	
nullableColumn_POSTGRE	2.01	1.99	-1.2010	1.46e-01	
preferUNIONALL_POSTGRE	2.00	1.97	-1.8493	9.93e-03	
redundantIndexes_POSTGRE	**264.87**	**12.61**	**-181.8202**	**6.34e-01**	
selectAll_POSTGRE	2.22	2.23	0.3657	2.58e-01	
sqlQueriesInsideLoop_POSTGRE	**5.37**	**1.99**	**-91.9668**	**6.86e-18**	**9**
tooManyIndexes_POSTGRE	5.43	5.36	-1.3439	5.46e-04	
useDedicatedStoredProcedure_POSTGRE	**2.52**	**2.07**	**-19.4555**	**6.86e-18**	
usingLONGDatatype_POSTGRE	3.13	3.97	23.4586	6.86e-18	
usingOldStyleJoin_POSTGRE	1.94	1.93	-0.5529	4.16e-01	
withoutPrimaryKey_POSTGRE	1.88	1.89	0.5216	3.24e-01	

All the patterns of *Expensive Calls in Loops* report an improvement when the violation is resolved, this reflect the substantial impact that loops have on inefficient code. Every pattern in this category is then included in the green smells set, except from *terminationExpressionInLoop*: even if it resolution brings a benefit in energy consumption, its value does not meet the necessary requirements, failing to exceed the 5% threshold.

Excluding the resulting green smells, the remaining set of patterns shows a negligible variation in the energy consumption: *impact* value vary from -4% to 3%. There are only 2 exceptions that experience a notable increase in the consumption. *largeNumberOfStringConcat* reports over 43% more power consumed, but its motivations were already explained before. *staticFieldCollection* also has 33.5% of additional required energy, this is caused by the behavior of the collection used in the pattern. Static HashMaps, that are permanently stored in memory, are tested against WeakHashMaps, that are implemented with weak references. A weak reference is a reference that is not strong enough to force an object to remain in memory, it leverages the ability of the garbage collector to determine references reachability instead of the program. This means that some sort of cleanup is periodically required: a removal of defunct entries of the collection id performed in order to avoid an ever-increasing number of dead WeakReferences. This behavior justifies the increment in energy consumption.

Unfortunately the tests relative to the Oracle management system presented several problems during the execution. Due to the high number of repetition required for generating a sample, the executions were occasionally slowed down, affecting the results of energy samplings. These results are then discarded from final evaluation. Only the values from MySQL and PostreSQL executions are taken into account. Table 5.2 reports the outcomes of energy measurement, marking the green smells in bold. The only patterns that are energetically inefficient in both the configurations are *existsIndipendentClause* and *sqlQueriesInsideLoop*. This result reflects the fact that for these two queries, a larger computational effort from Java is required. The computational load of the remaining set of patterns is evidently left to the database.

5.3 Correlation between green smells and execution time and resources usage

The second research question aims to find a correlation between the energy consumption and other variables: execution time and memory usage of the tested patterns. A description of the tests and an analysis of the obtained results gives an answer to the research question.

5.3.1 Experiment setup

Using the same setup of the previous tests, this two new analysis aims to detect improvements in performance time and in resource usage.

For what concert execution time, the pair of null and alternative hypothesis that are formulated for this test are:

$H2_0$: T_{with} and $T_{without}$ are similar

$H2_a$: T_{with} and $T_{without}$ are independent

where T is the execution time of the pattern under test, with or without the violation. Final decision is still taken according to the values of Mann-Whitney test using $\alpha = 0.05$ and to the impact on the overall time, in the same way as the previous test required. Patterns that satisfy the conditions are then named *performance smells*.

Table 5.3 Time results

method name	mean w	mean w/o	impact	p-value
callPropertiesThatCloneValuesInLoop	**2059.4**	**574.2**	**-112.784**	**6.74e-18**
indirectExceptionHandlingInsideLoops	**4539.5**	**936.3**	**-131.607**	**6.77e-18**
indirectStringConcatenationInLoops	**771.7**	**443.9**	**-53.928**	**6.48e-18**
instantiationInsideLoop	**1330.0**	**1126.5**	**-16.568**	**6.62e-18**
stringConcatenationInLoop	**3952.3**	**274.4**	**-174.032**	**6.60e-18**
terminationExpressionInLoop	620.3	616.9	-0.556	4.04e-03
boxedPrimitiveToString	**555.6**	**400.5**	**-32.431**	**6.82e-18**
boxingImmediatelyUnboxed	**39.0**	**30.9**	**-23.099**	**1.67e-13**
declareStaticMethod	**35.9**	**31.1**	**-14.153**	**6.38e-18**
dynamicInstantiation	**130.8**	**36.3**	**-113.085**	**5.99e-18**
instantiatingBoolean	**38.4**	**31.9**	**-18.658**	**1.52e-13**
largeNumberOfStringConcat	140.6	218.5	43.431	6.49e-18
needlessInstantiation	31.5	32.1	1.888	3.82e-01
newObjectForGetClass	**34.9**	**30.8**	**-12.530**	**7.59e-06**
numberCtor	**37.4**	**34.7**	**-7.379**	**6.88e-04**
randomUsedOnlyOnce	**7356.2**	**2006.0**	**-114.292**	**6.62e-18**
stringInitializationWithObject	**40.0**	**30.6**	**-26.856**	**1.20e-16**
callSuperFinalizeInFinally	21946.5	21785.4	-0.737	6.82e-18
callingFinalize	2159.4	2130.3	-1.358	6.31e-18
arrayCopy	985.5	974.9	-1.081	3.79e-08
collapsibleIfStatement	31.3	31.1	-0.577	7.06e-01
deadLocalStoreReturn	28.6	29.4	2.825	2.34e-01
formatStringNewLine	11446.1	11575.6	1.125	1.25e-02
localDoubleAssignment	29.4	30.2	2.552	4.12e-01
mutualExclusionOR	29.0	28.7	-0.970	5.48e-01
nonShortCircuit	**42.0**	**28.1**	**-39.589**	**5.59e-11**
repeatedConditionals	28.5	27.9	-1.844	2.99e-01
selfAssignment	29.1	29.0	-0.344	8.49e-01
staticFieldCollection	169.6	269.3	45.402	6.45e-18
switchRedundantAssignment	29.3	28.8	-1.789	4.14e-01
useArraysAsList	**968.6**	**105.2**	**-160.820**	**6.69e-18**
uselessControlFlow	28.8	29.4	1.926	3.81e-01
uselessSubstring	**34.3**	**30.7**	**-11.084**	**3.81e-05**
usingHashtable	**2814.1**	**1455.5**	**-63.638**	**6.75e-18**
usingRemoveAllToClearCollection	**857.7**	**555.7**	**-42.731**	**6.82e-18**
usingStringEmptyForStringTest	33.4	32.7	-2.239	2.16e-01
usingVector	752.8	733.7	-2.572	1.97e-14
variableDeclaredAsObject	455.6	455.2	-0.101	9.48e-01

Table 5.4 Time DB results

method name	mean w	mean w/o	impact	p-value
closeDBResourcesASAP_MYSQL	489	492	0.5707	3.63e-01
existsIndipendentClause_MYSQL	**1485**	**477**	**-102.6944**	**6.72e-18**
firstIndexInWhere_MYSQL	1893	1891	-0.1226	9.15e-01
implicitConversionInWHERE_MYSQL	502	498	-0.7518	2.70e-01
mixAnsiWithOracleSyntax_MYSQL	593	591	-0.3918	4.92e-01
nullableColumn_MYSQL	493	493	0.0243	9.23e-01
preferUNIONALL_MYSQL	491	489	-0.4078	5.12e-01
redundantIndexes_MYSQL	1935	1925	-0.5067	7.03e-02
selectAll_MYSQL	502	503	0.1712	8.82e-01
sqlQueriesInsideLoop_MYSQL	**4781**	**584**	**-156.4504**	**6.81e-18**
tooManyIndexes_MYSQL	7466	7288	-2.4221	7.96e-01
useDedicatedStoredProcedure_MYSQL	493	554	11.7301	6.72e-18
usingLONGDatatype_MYSQL	565	566	0.2900	5.32e-01
usingOldStyleJoin_MYSQL	549	545	-0.7862	1.37e-01
withoutPrimaryKey_MYSQL	478	473	-1.0091	1.06e-01
closeDBResourcesASAP_POSTGRE	259	259	-0.0464	6.59e-01
existsIndipendentClause_POSTGRE	**1254**	**257**	**-131.8584**	**6.71e-18**
firstIndexInWhere_POSTGRE	1546	1537	-0.5759	2.70e-01
implicitConversionInWHERE_POSTGRE	262	268	2.0530	8.74e-03
mixAnsiWithOracleSyntax_POSTGRE	257	259	0.8285	2.65e-01
nullableColumn_POSTGRE	257	259	0.4805	6.44e-01
preferUNIONALL_POSTGRE	259	260	0.3234	9.15e-01
redundantIndexes_POSTGRE	1626	1623	-0.1958	4.02e-01
selectAll_POSTGRE	267	261	-2.3130	1.44e-02
sqlQueriesInsideLoop_POSTGRE	**1106**	**283**	**-118.5693**	**6.79e-18**
tooManyIndexes_POSTGRE	1123	1110	-1.1931	4.75e-03
useDedicatedStoredProcedure_POSTGRE	**412**	**292**	**-34.0374**	**6.73e-18**
usingLONGDatatype_POSTGRE	390	502	25.1345	6.72e-18
usingOldStyleJoin_POSTGRE	255	258	1.4119	1.90e-01
withoutPrimaryKey_POSTGRE	250	252	0.9722	2.91e-01

The same approach is also used for the analysis of memory usage. A new couple of null and alternative hypothesis is now taken into account:

$H3_0$: M_{with} and $M_{without}$ are similar

$H3_a$: M_{with} and $M_{without}$ are indipendent

with M being the bytes of resources used by each pattern, with or without the violation. Memory allocation is calculated getting the free space left in the Java Virtual Machine. Results of this test are still evaluated according to the values of Mann-Whitney test and to the impact of total used bytes. The resulting patterns that pass all the thresholds are named *memory smells*.

5.3.2 Results

Time results are reported in Tables 5.3 and 5.4, indicating the execution time in millisecond for each pattern, and marking in bold the rows that satisfy all the requirements for being named performance smells.

In the same way, results of resource usage are reported in Tables 5.5 and 5.6, marking in bold the rows relative to the memory smells. It has to be noted that the results of this test are influenced by the behavior of the Garbage Collector: especially in "simpler" methods that do require smaller quantity of resources, no variation in the quantity of allocated bytes is registered. Some fields are then reported as blank, meaning that the variation impact is not reported.

Correlations between the results of different tests are reported in Tables 5.7 and 5.8. Each row has a mark that indicates whether the pattern is a green, a performance, or a memory smell. Even if those three sets of smells have a lot of patterns in common, it is evident that they do not coincide. More specifically, for what concern execution time correlation, *needlessInstantiation* does not presents performance improvements although being a green smell, and *uselessSubstring* on the contrary experience a faster execution that does not bring any energy saving. Similarly, memory consumption analysis shows that resources has correlations with the other tests: *largeNumberOfStringConcat*, *formatStringNewLine* and *staticFieldCollection* are reported to be memory smells but they actually do not affect neither energy consumption nor execution time.

5.4 Bytecode analysis

A final inspection on Java bytecode operations of the only patterns that are green smells is performed in order to detect those instruction that may be responsible of excessive energy consumption. Particular attention is given to the operations that handle methods invocation (`invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual`), the number of interactions with the stack (`push`, `load`, `pop` of values), and `new`, since these are the most common operations in the code.

`invokeinterface` is used to invoke a method declared within a Java interface. `invokespecial` can be used to invoke a private method of `this` Class, a method in a superclass of `this`, or principally to invoke the instance initialization method during the construction phase for a new object. `invokestatic` calls static methods, also known as class methods. Finally `invokevirtual` is used in Java to invoke all the methods that do not fall into the previous cases. `new` is used when creating object instances, it determines the size in bytes of the given class and allocates memory for the new instance from the garbage collected heap. A reference to the new object is also pushed onto the operand stack.

After the comparison of bytecode between both the element of every couple, it appears that the number of interactions with the stack does not affect the energy consumption. As reported in [56], stack operation consist in a constant small number of cycles, so they do not have a big impact. On the contrary, the main responsible in the computation are invoke-operations. Their execution requests a high number of cycles: `invokeinterface`, `invokespecial`, `invokestatic` and `invokevirtual` account respectively for 114+6r+l, 74+3*r+l, 74+3*r+l and 100+4r+l number of cycles, with r representing the constant for memory read and l the time required for loading into the cache. These specific instructions are among the ones that require most cycles to be performed, so it is clear that their execution requires a bigger computational effort by the processor, leading to a higher energy consumption. On the other hand, `new` execution time depends on the size of the created object: the amount of memory required for the allocation of the object is firstly filled with zeros, then instantiated.

When new instances are created, `invokespecial` and `new` are always executed: the first one is needed for allocating required memory for the new Object, the second for invoking the Class contstructor. It is clear now why a big part of green smells is related to useless Object instantiations: for the operations that it involves, it leads to a great energy spending.

The analysis of bytecode also supports the motivation behind the energy consumption increment caused by String concatenations. Due to the series of collateral operation that the + operator involves, for a single concatenation are executed a `new` for allocating memory for the StringBuilder object, an `invokespecial` for its constructor, and 3 `invokevirtual` for

appending the intermediate values and for the conversion of the object to a String. Using the `append` method on the other hand only involves a single `invokevirtual`.

Table 5.5 Memory results (in bytes)

method name	mean w	mean w/o	impact	p-value
callPropertiesThatCloneValuesInLoop	8.42e+07	2.60e+08	1.02e+02	2.22e-15
indirectExceptionHandlingInsideLoops	**5.53e+07**	**3.28e+07**	**-5.13e+01**	**1.51e-08**
indirectStringConcatenationInLoops	**3.53e+08**	**3.65e+07**	**-1.63e+02**	**6.07e-18**
instantiationInsideLoop	**3.31e+08**	**7.87e+06**	**-1.91e+02**	**2.38e-18**
stringConcatenationInLoop	**8.43e+07**	**1.35e+07**	**-1.45e+02**	**5.57e-18**
terminationExpressionInLoop	8.00e+07	8.00e+07	1.64e-03	3.85e-01
boxedPrimitiveToString	**8.57e+07**	**6.85e+07**	**-2.23e+01**	**4.61e-04**
boxingImmediatelyUnboxed	**4.43e+06**	**1.30e+06**	**-1.09e+02**	**2.70e-20**
declareStaticMethod	**3.16e+06**	**1.30e+06**	**-8.34e+01**	**1.10e-18**
dynamicInstantiation	2.99e+06	3.34e+06	1.11e+01	7.51e-01
instantiatingBoolean	**3.63e+06**	**1.30e+06**	**-9.45e+01**	**2.18e-20**
largeNumberOfStringConcat	2.23e+07	1.36e+07	-4.84e+01	5.89e-18
needlessInstantiation	1.33e+06	1.30e+06	-1.98e+00	1.55e-01
newObjectForGetClass	**3.43e+06**	**1.30e+06**	**-8.99e+01**	**2.36e-20**
numberCtor	**3.50e+06**	**1.30e+06**	**-9.16e+01**	**1.99e-20**
randomUsedOnlyOnce	1.81e+07	2.46e+07	3.05e+01	1.17e-04
stringInitializationWithObject	**5.36e+06**	**1.30e+06**	**-1.22e+02**	**2.74e-20**
callSuperFinalizeInFinally	1.60e+08	1.60e+08	3.42e-02	6.79e-18
callingFinalize	4.07e+08	4.07e+08	-1.06e-01	3.20e-20
arrayCopy	6.87e+07	6.89e+07	2.87e-01	3.04e-02
collapsibleIfStatement	1.30e+06	1.30e+06	-	-
deadLocalStoreReturn	1.30e+06	1.30e+06	-	-
formatStringNewLine	5.06e+06	3.66e+06	-3.21e+01	8.69e-11
localDoubleAssignment	1.30e+06	1.30e+06	-	-
mutualExclusionOR	1.30e+06	1.30e+06	-	-
nonShortCircuit	1.30e+06	1.30e+06	-	-
repeatedConditionals	1.30e+06	1.30e+06	-	-
selfAssignment	1.30e+06	1.30e+06	-	-
staticFieldCollection	1.40e+06	1.30e+06	-7.69e+00	3.35e-03
switchRedundantAssignment	1.30e+06	1.30e+06	-	-
useArraysAsList	**2.08e+08**	**1.39e+07**	**-1.75e+02**	**5.60e-18**
uselessControlFlow	1.30e+06	1.30e+06	-	-
uselessSubstring	1.30e+06	1.30e+06	-	-
usingHashtable	1.74e+08	3.19e+08	5.88e+01	6.57e-18
usingRemoveAllToClearCollection	**2.14e+08**	**1.77e+08**	**-1.92e+01**	**6.66e-18**
usingStringEmptyForStringTest	1.30e+06	1.30e+06	-	-
usingVector	4.17e+07	3.02e+08	1.51e+02	6.78e-18
variableDeclaredAsObject	6.76e+07	6.76e+07	-1.94e-03	2.33e-01

Table 5.6 Memory DB results

method name	mean w	mean w/o	impact	p-value
closeDBResourcesASAP_MYSQL	1.08e+07	10791602	-6.27e-02	4.65e-01
existsIndipendentClause_MYSQL	1.01e+07	10131273	4.34e-02	2.67e-12
firstIndexInWhere_MYSQL	5.43e+06	5426380	-1.47e-02	3.64e-01
implicitConversionInWHERE_MYSQL	1.14e+07	11436947	9.65e-02	1.76e-04
mixAnsiWithOracleSyntax_MYSQL	1.01e+07	10130430	4.97e-02	1.47e-13
nullableColumn_MYSQL	1.08e+07	10790641	2.19e-02	9.74e-13
preferUNIONALL_MYSQL	1.08e+07	10763796	3.30e-02	4.30e-05
redundantIndexes_MYSQL	2.69e+07	26839891	-1.23e-01	1.49e-02
selectAll_MYSQL	1.21e+07	12065276	3.65e-02	1.48e-01
sqlQueriesInsideLoop_MYSQL	**1.08e+07**	**10110629**	**-6.42e+00**	**7.29e-21**
tooManyIndexes_MYSQL	1.08e+07	10795160	1.55e-01	3.77e-17
useDedicatedStoredProcedure_MYSQL	1.06e+07	10466761	-1.62e+00	2.80e-06
usingLONGDatatype_MYSQL	3.41e+07	34150266	1.83e-01	7.20e-02
usingOldStyleJoin_MYSQL	1.01e+07	10110997	-3.72e-02	4.80e-14
withoutPrimaryKey_MYSQL	1.01e+07	10127365	-7.79e-02	9.44e-16
closeDBResourcesASAP_POSTGRE	7.33e+06	7334121	-2.62e-05	6.64e-01
existsIndipendentClause_POSTGRE	6.62e+06	6619864	7.54e-03	6.70e-20
firstIndexInWhere_POSTGRE	5.04e+06	5013852	-4.83e-01	3.44e-01
implicitConversionInWHERE_POSTGRE	7.33e+06	7333976	-3.60e-03	2.53e-23
mixAnsiWithOracleSyntax_POSTGRE	6.62e+06	6619361	9.67e-06	1.55e-01
nullableColumn_POSTGRE	7.33e+06	7334146	9.16e-04	1.95e-14
preferUNIONALL_POSTGRE	6.68e+06	6684014	2.54e-04	9.12e-02
redundantIndexes_POSTGRE	8.21e+06	8227905	2.58e-01	1.37e-04
selectAll_POSTGRE	8.53e+06	8531382	2.42e-03	1.06e-10
sqlQueriesInsideLoop_POSTGRE	**7.92e+06**	**5900482**	**-2.92e+01**	**1.06e-19**
tooManyIndexes_POSTGRE	7.19e+06	7194734	-7.81e-04	2.35e-02
useDedicatedStoredProcedure_POSTGRE	6.54e+06	6627550	1.26e+00	4.96e-20
usingLONGDatatype_POSTGRE	**3.90e+08**	**25668355**	**-1.75e+02**	**1.26e-16**
usingOldStyleJoin_POSTGRE	6.63e+06	6627608	3.40e-04	2.03e-21
withoutPrimaryKey_POSTGRE	6.62e+06	6619648	-2.24e-02	2.53e-23

Table 5.7 Correlations between green smells, performance smells and memory smells.

method name	energy smell	perf smell	mem smell
callPropertiesThatCloneValuesInLoop	×	×	
indirectExceptionHandlingInsideLoops	×	×	×
indirectStringConcatenationInLoops	×	×	×
instantiationInsideLoop	×	×	×
stringConcatenationInLoop	×	×	×
terminationExpressionInLoop			
boxedPrimitiveToString	×	×	×
boxingImmediatelyUnboxed	×	×	×
declareStaticMethod	×	×	×
dynamicInstantiation	×	×	
instantiatingBoolean	×	×	×
largeNumberOfStringConcat			×
needlessInstantiation	×		
newObjectForGetClass	×	×	×
numberCtor	×	×	×
randomUsedOnlyOnce	×	×	
stringInitializationWithObject	×	×	×
callSuperFinalizeInFinally			
callingFinalize			
arrayCopy			
collapsibleIfStatement			
deadLocalStoreReturn			
formatStringNewLine			×
localDoubleAssignment			
mutualExclusionOR			
nonShortCircuit	×	×	
repeatedConditionals			
selfAssignment			
staticFieldCollection			×
switchRedundantAssignment			
useArraysAsList	×	×	×
uselessControlFlow			
uselessSubstring		×	
usingHashtable	×	×	
usingRemoveAllToClearCollection	×	×	×
usingStringEmptyForStringTest			
usingVector			
variableDeclaredAsObject			

Table 5.8 Correlations between green smells, performance smells and memory smells, in database patterns.

method name	energy smell	perf smell	mem smell
closeDBResourcesASAP_MYSQL			
existsIndipendentClause_MYSQL	×	×	
firstIndexInWhere_MYSQL			
implicitConversionInWHERE_MYSQL			
mixAnsiWithOracleSyntax_MYSQL			
nullableColumn_MYSQL			
preferUNIONALL_MYSQL			
redundantIndexes_MYSQL			
selectAll_MYSQL			
sqlQueriesInsideLoop_MYSQL	×	×	×
tooManyIndexes_MYSQL			
useDedicatedStoredProcedure_MYSQL			
usingLONGDatatype_MYSQL			
usingOldStyleJoin_MYSQL			
withoutPrimaryKey_MYSQL			
closeDBResourcesASAP_POSTGRE			
existsIndipendentClause_POSTGRE	×	×	
firstIndexInWhere_POSTGRE			
implicitConversionInWHERE_POSTGRE			
mixAnsiWithOracleSyntax_POSTGRE			
nullableColumn_POSTGRE			
preferUNIONALL_POSTGRE			
redundantIndexes_POSTGRE	×		
selectAll_POSTGRE			
sqlQueriesInsideLoop_POSTGRE	×	×	×
tooManyIndexes_POSTGRE			
useDedicatedStoredProcedure_POSTGRE	×	×	
usingLONGDatatype_POSTGRE	×		×
usingOldStyleJoin_POSTGRE			
withoutPrimaryKey_POSTGRE			

5.5 Conclusions

This chapter described the setup and the results of the tests that were implemented in order to answer to the research question of the work. The answer to **RQ1** is: the detected green smells are 21: *callPropertiesThatCloneValuesInLoop, indirectExceptionHandlingInsideLoops, indirectStringConcatenationInLoops, instantiationInsideLoop, stringConcatenationInLoop, boxedPrimitiveToString, boxingImmediatelyUnboxed, declareStaticMethod, dynamicInstantiation, instantiatingBoolean, needlessInstantiation, newObjectForGetClass, numberCtor, randomUsedOnlyOnce, stringInitializationWithObject, nonShortCircuit, useArraysAsList, usingHashtable, usingRemoveAllToClearCollection, existsIndipendentClause* and *sqlQueriesInsideLoop*. Additional analysis on execution time, resources usage and bytecode were executed for answering the second research question. The answer to **RQ2** is: time and resource do not represent a cause in energy spending. On the contrary, bytecode instructions such as `invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual` and `new` have an impact on the consumption, since they require an higher number of cycles for their execution. `invokevirtual` and `new` are specifically executed during the creation of new instances, and then it explains why inefficient Objects instantiations are responsible for most of the green smells.

Chapter 6

Greenability Index

In this chapter the selected green smells are adapted in order to be included in the CAST analysis procedure. A custom quality index that measures the *Greenability*, i.e. the energy efficiency of sample applications, is then defined.

The first section illustrates in detail all the new rules and technical criteria that form the index, then in the second section the results of a sample analysis are shown.

6.1 Index implementation

CAST Assessment Model defines a set of rules that allows the generation of a Portfolio of Technical and Quality Assessment for given applications. The model is composed by a list of Quality Rules, that specify every flaw that can be present in applications under analysis. Every Rule belongs to one or more Technical Criteria, whose in turn belongs to one or more Business Criteria. Business Criteria can be either Health Factors (e.g. Changeability, Efficiency, Robustness, Security, Maintainability or Transferability) or Rule Compliance set (e.g. Architectural Design, Documentation and Programming Practices). When executed, the analysis assigns a final *grade* value from 4 to 0 to each Business Criterion, reflecting the quality of the application in relation to the Quality that the criterion represents. The calculation of the grade depends by its Quality Rules: each rule has a weight that indicates on the gravity of the violation it represents. Both the number of matches of these rules inside the application under analysis and their weights are taken into account by an internal algorithm for the computation of the final grade of the Technical Criterion they belongs to. In turn, each Technical Criterion has a weight, that in combination with its computed grade, determines the final grade of its Business Criterion.

The goal of this last part of the thesis is to extend the Assessment Model defining a new Health Factor named *greenability*, able to provide a quality assessment on the energy efficiency of given applications. This personalized Business Criterion is composed of all the green smells that were detected during the empirical analysis described in previous chapters. Some of this rules were already present in CAST Quality Rules set, so they are only modified in order to take part in the greenability measurement. All the other rules have to be implemented from scratch instead.

Implementation of customized Quality Rules involves a two steps process. The first part requires the creation of the appropriate knowledge base for every rule: this consist in the creation of a regular expression that can detect all the occurrences of the aforementioned rule. Once the Knowledge base is adapted, the second part involves the definition of the rule inside the Assessment Model. Alongside other various settings that regulate the rule behavior, this phase principally requires three settings:

- a documentation that describes the motivation and gives a practical example of the violation, reporting:
  - the technology it is referred to;
  - a brief description that presents the features a piece of code must satisfy in order to be reported by this rule;
  - a rationale explaining the reasons why this rule is recognized as “unhealthy”;
  - a remediation suggestion that shows how to correct the code so that the rule is solved and not detected anymore;
  - an optional application sample that gives a practical example on how the rule is triggered and how it is solved.

  Tables 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9 and 6.10 illustrate this step for every green smell that is not already present in CAST documentation, assigning a meaningful name to the rules.

- A weight value specify the importance that the violation has when computing the Business Criteria grade. This means that when a Rule is matched inside the analyzed application, if it has an high weight, it has a heavier impact on the final grade computation. On the contrary, a lighter weight impacts in a smaller way to the grade. Rules can be optionally marked as *critical*: this gives an additional gravity to the computation. When implementing the custom Rules, it was used the *rank* value that

was previously assigned during pattern testing, and it reflects the overall impact on energy consumption.

- Specification of the regular expression that matches the violation that was defined before.

Table 6.1 Division of green smells in Technical Criteria

section	method identifier
Expensive Calls in Loops	callPropertiesThatCloneValuesInLoop; indirectExceptionHandlingInsideLoops; indirectStringConcatenationInLoops; instantiationInsideLoop; stringConcatenationInLoop;
DB calls	existsIndipendentClause; sqlQueriesInsideLoop;
Expensive Object Instantiation	dynamicInstantiation; instantiatingBoolean; stringInitializationWithObject; declareStaticMethod; boxingImmediatelyUnboxed; numberCtor; randomUsedOnlyOnce; needlessInstantiation; boxedPrimitiveToString; newObjectForGetClass;
Poor Programming Choices	nonShortCircuit; useArraysAsList; usingHashtable; usingRemoveAllToClearCollection;

Once each rule is defined, a following step requires to assign each rule to the belonging Technical Criterion. The purpose of these Criteria is to assemble all those violation that belong to the same semantic field. For this purpose it is used the same division in categories that was already used in previous chapters, as Table 6.1 reports. The four specified Technical Criteria, Expensive Calls in Loops, DB Calls, Expensive Object Instantiation and Poor Programming Choices, they all brings a contribution to the computation of the Greenability final value. When all the rules and the dependencies between them, the Technical and Business Criteria are defined, the code analysis is automatically performed by CAST, and the final grade are computed by an internal weighted algorithm and finally displayed in the CAST AIP.

Table 6.2 documentation of boxingImmediatelyUnboxed green smell

Avoid boxing a primitive value and then immediately unbox it to perform primitive coercion	
description	A primitive boxed value is constructed and then is immediately converted into a different primitive type.
rationale	A useless intermediate object is created
resolution	A better and easier solution is to just perform the direct primitive coercion.
sample	`new Double(d).intValue();`
remediation sample	`(int) d;`

Table 6.3 documentation of numberCtor green smell

Avoid inefficient Number constructor calls	
description	Reports all the invocation of the Number classes (`Integer`, `Long`, `Short`, `Character`, `Byte`) constructor.
rationale	Using the constructor results in the instantiation of a new object, whereas is possible to make it to be done by the compiler, JVM or the class library using cached values. For values between -128 and 127 better performances are also guaranteed.
resolution	Use the `valueOf()` method to use cached values.
sample	`new Integer(int);`
remediation sample	`Integer.valueOf(int);`

Table 6.4 documentation of randomUsedOnlyOnce green smell

Never create a new Random object for each invocation and use it only once	
description	This rule reports the code that creates a `java.util.Random` object, uses it to generate one random number, and then immediately discards the `Random` object.
rationale	This solution is poorly efficient and produce mediocre quality random numbers, resulting in easily guessable values.
resolution	If possible, create the `Random` object once, save it and reuse it without recreating it every time. Otherwise, consider using a `java.security.SecureRandom` object avoiding its reallocation everytime a new value is required.

sample	`public int extract() {`
	`int x = new Random().nextInt();`
	`return x;`
	`}`

remediation sample	`Random Rand = new Random();`
	
	`public int extract() {`
	`int x = Rand.nextInt();`
	`return x;`
	`}`

Table 6.5 documentation of usingRemoveAllToClearCollection green smell

Avoid using the removeAll() method for clearing a collection	
description	It reports all the use of `collection.removeAll(collection)` for clearing a collection.
rationale	The use of this method leads to poor readability, less efficient and used with some collection (like `List`) might trhow a `ConcurrentModificationException`
resolution	Use `collection.clear()` instead.
sample	`collection.removeAll(collection);`
remediation sample	`collection.clear();`

Table 6.6 documentation of needlessInstantiation green smell

Classes that only supply static methods should never be instantiated	
description	This rule looks for the allocation of objects that are based on a class that only supplies static methods.
rationale	The allocated object does not need to be created and is useless since the class does not need to be instantiated.
resolution	Just access the static methods directly using the class name as a qualifier.

sample	`MyClass c = new MyClass();`
	`c.foo();`
	`...`
	`Class MyClass {`
	`public static void foo(){`
	`//do stuff`
	`}`
	`}`

remediation sample	`MyClass.foo();`
	`...`
	`Class MyClass {`
	`public static void foo(){`
	`//do stuff`
	`}`
	`}`

Table 6.7 documentation of boxedPrimitiveToString green smell

Avoid allocating a boxed primitive value just to call toString()	
description	This rule is triggered when a boxed primitive is allocated just for calling the `toString()` method.
rationale	The object instantiation is a useless operation that wastes resources.
resolution	It is more effective to just use the static form of `toString` which takes the primitive value.

sample	`new Integer(1).toString()`
	`new Long(1).toString()`
	`new Float(1.0).toString()`
	`new Double(1.0).toString()`
	`new Byte(1).toString()`
	`new Short(1).toString()`
	`new Boolean(true).toString()`

remediation sample	`Integer.toString(1)`
	`Long.toString(1)`
	`Float.toString(1.0)`
	`Double.toString(1.0)`
	`Byte.toString(1)`
	`Short.toString(1)`
	`Boolean.toString(true)`

Table 6.8 documentation of newObjectForGetClass green smell

Avoid allocating an object just to call getClass() method	
description	This code allocates an object just to call `getClass()` on it, in order to retrieve the `Class` object for it.
rationale	It is a useless allocation of a `Class` instance.
resolution	It is simpler to just access the `.class` property of the class.
sample	`Integer i = Integer.valueOf(11);` `Class c = i.getClass();`
remediation sample	`Class c = Integer.class;`

Table 6.9 documentation of useArraysAsLists green smell

Never create a list from an array copying its element with a loop	
description	This rule is activated every time a list is populated copying the element of an array of Objects one by one through a loop.
rationale	Looping through an array can result slow and poorly efficient.
resolution	The `java.util.Arrays` class has a `asList` method that should be used when you want to create a new List from an array of objects.

sample	`Integer ints[] = new Integer[];`
	
	`List l = new ArrayList();`
	`for (int i=0; i<ints.length; i++) {`
	`l.add(ints[i]);`
	`}`

remediation sample	`Integer ints[] = new Integer[20];`
	
	`List l = Arrays.asList(ints);`

Table 6.10 documentation of nonShortCircuit green smell

Avoid using non short circuit operators in boolean conditions	
description	This rule detect the usage of non-short-circuit logic (e.g., & or |) rather than short-circuit logic (&& or ||).
rationale	Non-short-circuit logic causes both sides of the expression to be evaluated even when the result can be inferred from knowing the left-hand side. This can be less efficient and can result in errors if the left-hand side guards cases when evaluating the right-hand side can generate an error.
resolution	Use short-circuit logic operators.

sample	`int x = 4;`
	`int y = 2;`
	
	`if ( x > 5 & x / y > 1 ) {`
	`// do stuff`
	`}`

remediation sample	`int x = 4;`
	`int y = 2;`
	
	`if ( x > 5 & x / y > 1 ) {`
	`// do stuff`
	`}`

6.2 Practical application of greenability measurement

After the implementation of the greenability index, a practical evaluation on a Java application is performed. The sample analysis uses WebGoat[1], a deliberately flawed application that allows interested developers to test vulnerabilities commonly found in Java-based applications.

After the analysis is performed, the CAST Engineering Dashboard shows for each Business and Technical Criterion, every occurrence of the violated Rules. Every Business and Technical Criterion as well as every rule has a grade from 4 to 0, reflecting the gravity of the metric under analysis present in current application. In Figure 6.1, webgoat analysis results are divided into different Business Criteria; greenability is highlighted alongside its corresponding Technical Criteria: Expensive Calls in Loops, Expensive Object Instantiations, DB Calls and Poor Programming Choices. The overall value of webgoat greenability is 3.58, indicating an acceptable level of energy efficiency, while the less efficient Technical Criteria is *Expensive Calls in Loops*, with a grade of 2.68. Figure 6.2 illustrates all the Quality Rules taken into account for the computation of the Expensive Object Instantiation criteria, also reporting their grade. For every detected rule, the Dashboard also points the portion of code that triggers the violation, highlighted in red (Figure 6.3).

The implementation of this index offers a useful methodology for assessing sample Java applications in order to evaluate their energy efficiency, and gives the ability to analyze and refactor the code from a energy efficient point of view. Even if containing a limited number of violations, it represents a starting point for making the users aware of power management of their software environments.

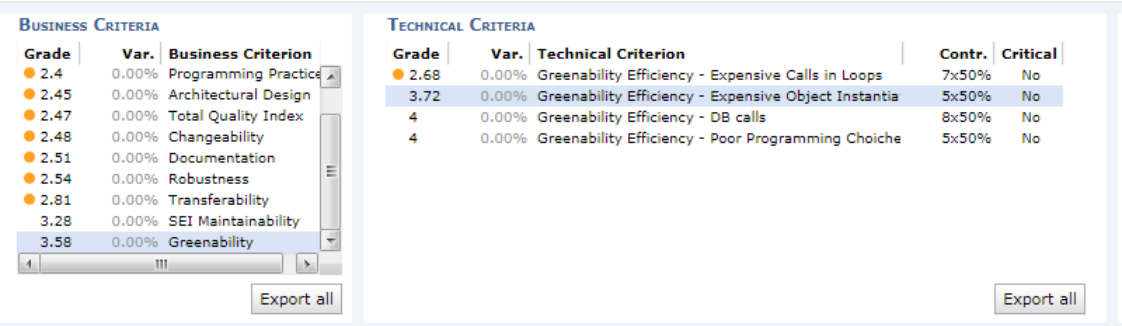
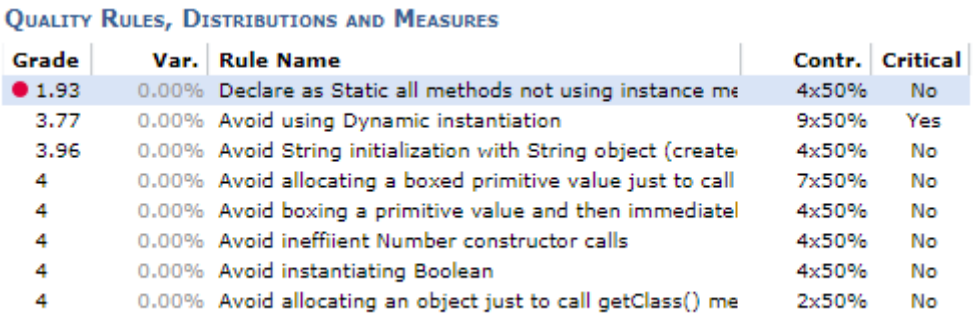


Fig. 6.1 Business Criteria that are taken into account for webgoat analysis, with greenability metric and its Technical Criteria.

[1]https://github.com/WebGoat/WebGoat/wiki



QUALITY RULES, DISTRIBUTIONS AND MEASURES

Grade	Var.	Rule Name	Contr.	Critical
1.93	0.00%	Declare as Static all methods not using instance me	4x50%	No
3.77	0.00%	Avoid using Dynamic instantiation	9x50%	Yes
3.96	0.00%	Avoid String initialization with String object (create	4x50%	No
4	0.00%	Avoid allocating a boxed primitive value just to call	7x50%	No
4	0.00%	Avoid boxing a primitive value and then immediatel	4x50%	No
4	0.00%	Avoid ineffiient Number constructor calls	4x50%	No
4	0.00%	Avoid instantiating Boolean	4x50%	No
4	0.00%	Avoid allocating an object just to call getClass() me	2x50%	No

Fig. 6.2 Quality Rules belonging to Expensive Object Instantiations Criterion.

```
        if (lesson != null)
        {
            source = lesson.getSource(s);
        }
    }
    if (source == null)
    {
        return "Source code is not available. Contact webgoat@g2-inc.com";
    }
    return (source.replaceAll("(?s)" + START_SOURCE_SKIP + ".*"
            + END_SOURCE_SKIP, "Code Section Deliberately Omitted"));
}


/**
 *  Description of the Method
 *
 * @param  s                Description of the Parameter
 * @param  response         Description of the Parameter
 * @exception  IOException  Description of the Exception
 */
protected void writeSource(String s, HttpServletResponse response)
        throws IOException
{
    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

    if (s == null)
    {
        s = new String();
    }

    out.print(s);
    out.close();
}
```

Fig. 6.3 A sample occurrence of *Avoid String Initialization with String object* violation inside the code.

This chapter illustrated the implementation of a quality index that evaluated the greenability of sample applications. Using CAST Assessment Model, every green smell was implemented as Quality Rule, and then assigned to the belonging Technical Criteria. A practical example

was given with the analysis of webgoat application, reporting the final value of the index and the impact grade of every rule.

Chapter 7

Conclusions and future works

Improvements in the energy management of IT system can lead to great financial savings and greenhouse gases emission reductions. The principal responsible in energy consumption are both the hardware components, that physically require power in order to operate, and software instructions, that specify the hardware behavior. Main goal of this work is to evaluate the impact that specific patterns in Java code have on the total power consumption of applications, in order to define a set of bad-practices for a more environmental friendly product. Starting from a set of 53 identified patterns, a series of tests measured the difference in energy consumption between the methods that included the violation and their respective "healed" counterpart. This set of patterns was detected through the analysis of well known violations related to efficiency, performance, robustness, bad practices and dodgy code rules, reported in the documentation of CAST Quality Rules and other Automatic Static Analysis tools.

The aim of the thesis was summarized using two research question:

RQ1: which of the patterns under test represent an energy hotspot (i.e. which are green smells)?

- After the tests, a final list of 21 patterns were proved to be energy inefficient, and they were therefore defined as *green smells*. Every smell is assigned to a category of similar smells, and a *rank* value is used for describing the impact that it has on the overall consumption. Most common inefficiencies are related to non-optimal Object instantiations and to expensive operation executed inside loops.

RQ2: which are the main variables between execution time, resource usage and bytecode instruction, that are responsible of the increase in energy consume?

- Research Question 2 was intended to find those variables that may be the cause of the energy consumption increment. Tests were rerun firstly considering the execution time as variable under analysis, then resource usage; this permitted to define two new sets of *performance smells* and *resource smells*. Results show no direct effect-cause relationship between energy and any of the measurements, since neither performance smells nor resource smells sets coincide with the green smells group. A last possible answer to RQ2 was explored in Java bytecode operations, in order to detect those instructions that may influence energy consumption. This analysis highlighted a higher distribution of principally two instructions: `new` and *invoke*-methods operations were present in higher number inside violation implementations rather then in their healed versions. This results is motivated by the fact that this pair of instructions is principally required during Objects instantiation: `new` for allocating memory for the new instance and `invokespecial` for calling the Class constructor. This indicates auxiliary Objects created at runtime as one of the main causes for energy waste.

In the final phase of the thesis, a custom quality index that measures the *greenability*, i.e. the energy efficiency of sample applications, is defined. Using CAST Application Intelligence Platform, all the detected smells were implemented as Quality Rules and inserted into the computation for the metric value. Execution on a sample application proved the goodness of the Index.

This thesis was intended to detect an initial set of green smells, in order to define a methodology to assess energy efficiency of software applications. Using current results, future works can focus on expanding the research for new energy hotspots, focusing on inefficient Object instantiations.

An interesting approach would be the comparison of energy consumption according to the presence of Fowler [14] code smells. More precisely, confrontations of particular smells like *Large classes* or *Long methods*, in contrast with others like *Feature envy* or *Lazy classes* can lead to significant results. First ones indeed can indicate a reduction in Object instantiations and methods invocations, leading to energy saving. On the contrary the second group of smells reflects a bad use of class resources, this may increase the number of *invoke*-methods in order to access to external features thus increasing power spending.

It should also important to better investigate the impact that specific bytecode operations has on the energy spending. A removal of the `invokespecial` instruction is for example possible through the decapsulation of a Class varibles. Even if against Object Oriented

Programming practices, an analysis on the effects that this specific instruction has on the consumtpion could be significant.

This example also introduce another subject: improving energy efficiency of software applications through, for example, the detection and correction of green smells, can raise possible trade-off choices. Since energy smells not always are also performance smells, in some situations a greener solution can severely impact the execution time of the applications, and vice versa. Further studies should then quantify the benefits that a solution can give in opposition to the other, finding, if it possible, which are the best trade-off solutions.

Bibliography

[1] Susanne Albers and Antonios Antoniadis. Race to idle: new algorithms for speed scaling with a sleep state. *ACM Transactions on Algorithms (TALG)*, 10(2):9, 2014.

[2] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[3] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12), 2007.

[4] Marco Bessi. A methodology to improve the energy efficiency of software. 2014.

[5] Gale Boyd, Elizabeth Dutrow, and Walt Tunnessen. The evolution of the energy star® energy performance indicator for benchmarking industrial plant manufacturing energy use. *Journal of cleaner production*, 16(6):709–715, 2008.

[6] Richard E Brown, Richard Brown, Eric Masanet, Bruce Nordman, Bill Tschudi, Arman Shehabi, John Stanley, Jonathan Koomey, Dale Sartor, Peter Chan, et al. Report to congress on server and data center energy efficiency: Public law 109-431. Technical report, Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), 2007.

[7] Christian Bunse, Hagen Höpfner, Essam Mansour, and Suman Roychoudhury. Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In *Mobile Data Management: Systems, Services and Middleware, 2009. MDM'09. Tenth International Conference on*, pages 600–607. IEEE, 2009.

[8] Eugenio Capra, Chiara Francalanci, and Sandra A Slaughter. Is software "green"? application development environments and energy efficiency in open source applications. *Information and Software Technology*, 54(1):60–71, 2012.

[9] Eugenio Capra and Francesco Merlo. Green it: Everything starts from the software. 2009.

[10] S Chu. The energy problem and lawrence berkeley national laboratory. *Talk given to the California Air Resources Board*, 2008.

[11] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194. IEEE, 2010.

[12] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 13–23. ACM, 2007.

[13] Jason Flinn and Mahadev Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop on*, pages 2–10. IEEE, 1999.

[14] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[15] Rita Garcia and Fausto Freire. Carbon footprint of particleboard: a comparison between iso/ts 14067, ghg protocol, pas 2050 and climate declaration. *Journal of Cleaner Production*, 66:199–209, 2014.

[16] Marion Gottschalk, Jan Jelschen, and Andreas Winter. Saving energy on mobile devices by refactoring. In *EnviroInfo*, pages 437–444, 2014.

[17] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing energy code smells with reengineering services. *GI-Jahrestagung*, 208:441–455, 2012.

[18] I Greenpeace. Make it green: Cloud computing and its contribution to climate change, 2010.

[19] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[20] Anna Hart. Mann-whitney test is not just a test of medians: differences in spread can be important. *BMJ: British Medical Journal*, 323(7309):391, 2001.

[21] Scott Hemmert. Green hpc: From nice to necessity. *Computing in Science & Engineering*, 12(6):8–10, 2010.

[22] Erik A Jagroep, Jan Martijn van der Werf, Sjaak Brinkkemper, Giuseppe Procaccianti, Patricia Lago, Leen Blom, and Rob van Vliet. Software energy profiling: Comparing releases of a software product. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 523–532. ACM, 2016.

[23] Jan Jelschen, Marion Gottschalk, Mirco Josefiok, Cosmin Pitu, and Andreas Winter. Towards applying reengineering services to energy-efficient applications. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 353–358. IEEE, 2012.

[24] S Josselyin, B Dillon, M Nakamura, R Arora, S Lorenz, T Meyer, R Maceska, and L Fernandez. Worldwide and regional server 2006-2010 forecast. *IDC report, November*, 2006.

[25] Eva Kern, Markus Dick, Stefan Naumann, and Tim Hiller. Impacts of software and its engineering on the carbon footprint of ict. *Environmental Impact Assessment Review*, 52:53–61, 2015.

[26] David HK Kim, Connor Imes, and Henry Hoffmann. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2015 IEEE 3rd International Conference on*, pages 78–85. IEEE, 2015.

[27] Volodymyr Kindratenko and Pedro Trancoso. Trends in high-performance computing. *Computing in Science & Engineering*, 13(3):92–95, 2011.

[28] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, 9, 2011.

[29] Jonathan G Koomey. Outperforming moore's law. *IEEE Spectrum*, 47(3):68–68, 2010.

[30] John A Laitner and Mike Berners-Lee. Gesi smarter 2020: The role of ict in driving a sustainable future. Technical report, Technical report, Global e-Sustainability Initiative, 2012.(Cited on page 2.), 2012.

[31] Luís Gabriel Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, Gilberto Melfe, and João Paulo Fernandes. Haskell in green land: Analyzing the energy behavior of a purely functional language. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 517–528. IEEE, 2016.

[32] Kenan Liu, Gustavo Pinto, and Yu David Liu. Data-oriented characterization of application-level energy optimization. In *FASE*, pages 316–331, 2015.

[33] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[34] Joanna McGrenere. Bloat: the objective and subject dimensions. In *CHI'00 Extended Abstracts on Human Factors in Computing Systems*, pages 337–338. ACM, 2000.

[35] David Meisner, Brian T Gold, and Thomas F Wenisch. Powernap: eliminating server idle power. In *ACM Sigplan Notices*, volume 44, pages 205–216. ACM, 2009.

[36] Davide Morelli, Andrea Canciani, and Antonio Cisternino. A high-level and accurate energy model of parallel and concurrent workloads. *Concurrency and Computation: Practice and Experience*, 28(3):822–833, 2016.

[37] Lev Mukhanov, Pavlos Petoumenos, Zheng Wang, Nikos Parasyris, Dimitrios S Nikolopoulos, Bronis R De Supinski, and Hugh Leather. Alea: a fine-grained energy profiling tool. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(1):1, 2017.

[38] San Murugesan. Harnessing green it: Principles and practices. *IT professional*, 10(1), 2008.

[39] Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. Runtime monitoring of software energy hotspots. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 160–169. ACM, 2012.

[40] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Unit testing of energy consumption of software libraries. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1200–1205. ACM, 2014.

[41] Murray G Patterson. What is energy efficiency?: Concepts, indicators and methodological issues. *Energy policy*, 24(5):377–390, 1996.

[42] Steven Pelley, David Meisner, Thomas F Wenisch, and James W VanGilder. Understanding and abstracting total data center power. In *Workshop on Energy-Efficient Design*, 2009.

[43] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages. 2017.

[44] G Pinto and F Castor. Characterizing the energy efficiency of java's thread-safe collections in a multicore environment. In *Proceedings of the SPLASH'2014 workshop on Software Engineering for Parallel Systems (SEPS), SEPS*, volume 14, 2014.

[45] Gustavo Pinto and Fernando Castor. On the implications of language constructs for concurrent execution in the energy efficiency of multicore applications. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, pages 95–96. ACM, 2013.

[46] 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*, pages 22–31. ACM, 2014.

[47] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *ACM SIGPLAN Notices*, volume 49, pages 345–360. ACM, 2014.

[48] Giuseppe Procaccianti, Luca Ardito, Maurizio Morisio, et al. Profiling power consumption on desktop computer systems. In *International Conference on Information and Communication on Technology*, pages 110–123. Springer, 2011.

[49] Giuseppe Procaccianti, Stefano Bevini, Patricia Lago, et al. Energy efficiency in cloud software architectures. 2013.

[50] Giuseppe Procaccianti, Hector Fernandez, and Patricia Lago. Empirical evaluation of two best practices for energy-efficient software development. *Journal of Systems and Software*, 117:185–198, 2016.

[51] Giuseppe Procaccianti, Patricia Lago, and Grace A Lewis. A catalogue of green architectural tactics for the cloud. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2014 IEEE 8th International Symposium on the*, pages 29–36. IEEE, 2014.

[52] Philip E Ross. 5 commandments [technology laws and rules of thumb]. *IEEE Spectrum*, 40(12):30–35, 2003.

[53] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *Ieee micro*, 32(2):20–27, 2012.

[54] Snehanshu Saha, Jyotirmoy Sarkar, Avantika Dwivedi, Nandita Dwivedi, Anand M. Narasimhamurthy, and Ranjan Roy. A novel revenue optimization model to address the operation and maintenance cost of a data center. *Journal of Cloud Computing*, 5(1), Jan 2016.

[55] Manasa Sahini. *Experimental and computational study of multi-level cooling systems at elevated coolant temperatures in data centers*. PhD thesis, 2017.

[56] Martin Schoeberl. *Jop: A java optimized processor for embedded real-time systems*. VDM Publishing, 2008.

[57] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. United states data center energy usage report. 2016.

[58] Sanath S Shenoy and Raghavendra Eeratta. Green software development model: An approach towards sustainable software development. In *India Conference (INDICON), 2011 Annual IEEE*, pages 1–6. IEEE, 2011.

[59] Anne E Trefethen and Jeyarajan Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 4(6):444–449, 2013.

[60] Roberto Verdecchia, Giuseppe Procaccianti, Ivano Malavolta, Patricia Lago, and Joost Koedijk. Estimating energy impact of software releases and deployment strategies: The kpmg case study. In *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on*, pages 257–266. IEEE, 2017.

[61] Antonio Vetro, Luca Ardito, Maurizio Morisio, and Giuseppe Procaccianti. Monitoring it power consumption in a research center: Seven facts. 2011.

[62] Antonio Vetro, Luca Ardito, Giuseppe Procaccianti, and Maurizio Morisio. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. 2013.

[63] Claas Wilke, Sebastian Richly, Sebastian Gotz, Christian Piechnick, and Uwe Aßmann. Energy consumption and efficiency in mobile applications: A user feedback study. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 134–141. IEEE, 2013.

[64] Tomofumi Yuki and Sanjay Rajopadhye. Folklore confirmed: Compiling for speed = compiling for energy. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 169–184. Springer, 2013.