Tales from the Code #1: The Effective Impact of Code Refactorings on Software Energy Consumption

Zakaria Ournani[1,2,3], Romain Rouvoy[2,3], Pierre Rust[1] and Joel Penhoat[1]
[1] *Orange Labs, Rennes, France*
[2] *INRIA Lille Nord-Europe, Lille, France*
[3] *University of Lille, Lille, France*

Keywords: Code Refactoring, Empirical Software Engineering, Software Energy Consumption.

Abstract: Software maintenance and evolution enclose a broad set of actions that aim to improve both functional and non-functional concerns of a software system. Among the non-functional concerns, energy consumption is getting more and more traction in the industry, no matter the software is mobile or deployed in the cloud. In this context, the impact of code refactorings on energy consumption remains unclear, though. In particular, while the state of the art investigated the impact of some specific code refactorings on dedicated benchmarks, we miss an assessment that those apply to more comprehensive and complex software. To address this threat, this paper studies the evolution of the energy consumption of 7 open-source software developed for more than 5 years. Then, by focusing on the impact on energy consumption of changes involving code refactorings, we intend to assess the effects induced by such code refactorings in practice. For all these software systems we studied, our empirical results report that the code refactorings we mined do not substantially impact energy consumption. Interestingly, these results highlight that *i)* structural code refactorings bring energy-preserving changes to the code, and *ii)* major energy variations seem to be related to functional and computational code evolutions.

1 INTRODUCTION

Software energy consumption has gained a substantial significance in the last decade, both for research and industrial contexts (Verdecchia et al., 2017; Pinto et al., 2016; Rodriguez, 2017; Chowdhury et al., 2019; Fonseca et al., 2019). Hence, many researchers and practitioners started caring about the energy efficiency of software, beyond performance and hardware concerns (Cruz et al., 2017; Pinto et al., 2014; Manotas et al., 2016; Manotas et al., 2013). Being integrated into mobile or cloud environments, software systems are trying to minimize their resource consumption to reduce battery consumption or operational cost.

In this context, the impact of software development techniques on energy consumption has been explored by the state of the art—including code compilation, static code analysis, code refactorings—which is the focus of this paper. Source code refactorings can be described as the application of acknowledged rules to improve one or many aspects of a software system, such as its clarity, maintenance, code smells, without impacting its functional behavior (Kerievsky, 2004; Abid et al., 2020).

Yet, code refactorings have also been considered as a mean to improve the performance and/or energy efficiency in a more or less automated way (Gottschalk et al., 2013; Anwar et al., 2019; Cruz et al., 2017; Morales et al., 2018; Cruz and Abreu, 2017; Bree and Cinnéide, 2020). The large majority of the literature that has been published in this domain—especially for mobile application (Palomba et al., 2019; Gottschalk et al., 2013; Anwar et al., 2019; Linares-Vásquez et al., 2014)—based their study on a predefined set of refactoring rules, design patterns, or code smells. In most of these studies, the authors measure and analyze the effect of atomic code changes on the total energy efficiency of the software under study, before concluding on their effect. While this process may deliver interesting insights on the impact of specific code refactorings on the energy consumption of a code snippet, there is still no guarantee that the identified code refactorings are frequently applied during the lifespan of a software system. Some refactorings could be very advantageous

Ournani, Z., Rouvoy, R., Rust, P. and Penhoat, J.
Tales from the Code 1: The Effective Impact of Code Refactorings on Software Energy Consumption.
DOI: 10.5220/0010517900340046
In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*, pages 34-46
ISBN: 978-989-758-523-4

but are rarely applied which limits their impact on the energy efficiency of the software.

In this paper, we thus consider an alternative approach to study the impact of code refactorings on the energy efficiency of legacy software systems. We focus on acknowledged refactoring rules mostly issued from Martin Fowler's book (Fowler, 1999), which are mostly structure-oriented rules (such as Extract Method) dealing with code architecture and organization for server-side applications rather than implementation and computation changes (such as Substitue Algorithm). Instead of selecting a set of code refactorings *a priori* and evaluate them against some dedicated benchmarks, we extract these code refactorings from established open-source projects. More specifically, we mine the history of code refactorings that have been applied to these projects in the past, and we measure the impact of the commits that include acknowledged code refactorings on the overall energy consumption. This approach aims to detect the code refactorings that have been broadly applied, and their observable impact on energy efficiency in practice. By doing so, we believe that mined code refactorings are most likely to reflect an effective impact of code refactoring on energy consumption, compared to the study of a fixed set of refactoring candidates. This study, therefore, aims to answer the following research questions:

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

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

Furthermore, beyond answering these two questions, the paper comes with a set of contributions that can be summarized as:

1. Proposing a new empirical approach to study the impact of source code refactorings on the energy consumption of software systems,
2. Investigating the contribution of code refactorings to the global evolution of software energy consumption,
3. Providing a detailed description of the most applied code refactorings and their impact on energy consumption,
4. Validating the code refactoring effects on energy consumption through statistical tests and micro-benchmarking.

The remainder of this paper is organized as follows. Section 2 introduces the experimental protocol (hardware, projects, tools, and methodology) we adopted in this study. Section 3 analyzes several experiments we conducted to mine the code refactorings and evaluate their impact on the energy consumption, as well as the results we observed during these experiments. Section 4 discusses the related work about source code refactoring contributions to reduce software energy consumption. Finally, Sections 5 and 6 cover the validity threats and our conclusions, respectively.

2 EXPERIMENTAL PROTOCOL

This section describes our detailed experimental environment, encompassing the hardware configuration, the studied projects/benchmarks and a detailed description of our experimental methodology.

2.1 Hardware Environment

For all of our experiments, we used a Core i7 machine (i7-6600U CPU @ 2.60GHz) with a total of 4 processing units to measure the energy consumption and mine the refactoring rules from the projects under study. The machine ran a 18.04.4 LTS Ubuntu, with a `4.15.0-88-generic` Linux kernel. We also used OPENJDK, version 1.8.0_242, to run most of our Java experiments—*i.e.*, run both old and recent versions—except for the OkHttp project where we had to use OPENJDK, version 11.0.6. By using the same machine to conduct all the experiments, we guarantee the least energy consumption variation and a controlled impact of the hardware configuration, as recommended in (Ournani et al., 2020), especially to measure small differences in energy consumption. Therefore, the machine has been configured according to guidelines of (Ournani et al., 2020) to mitigate the energy consumption variation to the minimum and produce accurate results that can be faithfully compared.

2.2 Projects under Study

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

Table 1: List of selected open-source projects.

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

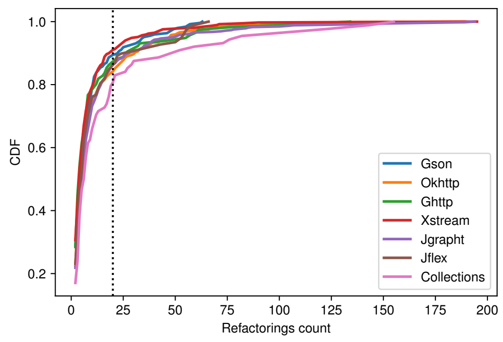


Figure 1: CDF of code refactorings per commit.

interface, and in which the main functions are non-ambiguously identified, so we can run the same measurements across different generations and versions of the studied projects.

Based on the above criteria, Table 1 summarizes the projects that we considered for this study, along with the number of commits at the time that this paper was written, and the date of the first commit. Established projects with a higher number of commits increase the chances to mine a representative set of commits including code refactorings. All the projects we selected have been hosted on GitHub since at least 2015. We note that the Git creation date only gives an overview of how long has the project been on GitHub and is different from the project creation date. Some projects, such as Gson, exists on GitHub since March 2015, but we still can checkout commits from 2003.

2.3 Methodology & Tools

Our experimental methodology is a process that includes extraction, evaluation, and validation steps. Figure 2 depicts the main steps we followed to analyze each selected project. We start our process by cloning the public repository of the project from GitHub. Then, for each commit, we mine the code refactorings of the project using the REFACTORINGMINER tool and we summarize them into a JSON file. REFACTORINGMINER is an open-source research project (Tsantalis et al., 2018; Tsantalis et al., 2020) that analyses a project commit by commit and extracts the type and count of refactorings for each commit in a JSON format. It helps in detecting and visualizing 55 different types of refactoring in its version 2.0, which is the version we used in this study.[1]

Once we extract the code refactorings that have been applied per commit on the master branch, we select the commits to be reproduced to measure their energy consumption. The selection method takes into account the refactorings count and types in each commit. We consider commits with at least 20 refactorings so we can expect a significant impact of the refactorings on the energy consumption. Figure 1 depicts the *cumulative distribution function* (CDF) that shows the frequency of commits per refactorings count (commits with more than 200 refactorings have been omitted for clarity). For most of the studied projects, one can see that 20% of the commits have more than 20 refactorings. This ensures having a decent number of the most relevant commits that can be reproduced and evaluated. The commits that contain only one type of refactoring are very rare, we thus also consider commits with a mix of code refactorings, and deduce the impact of each refactoring rule *a posteriori*.

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

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

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

mutable and immutable collections (lists, maps, sets), inspired from (Pinto et al., 2016; Samir Hasan et al., 2016). Using JMH for writing benchmarks has many advantages, such as the easy management of run and warm-up iterations, and the prevention of dead code removal from the JIT using the concept of *black-hole* (Rodriguez-Cancio et al., 2016).

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

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

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

During the execution of the experiments, we use Intel RAPL to acquire the global energy consumption (Khan et al., 2018; Desrochers et al., 2016), which is one of the most accurate available tools to report the CPU/DRAM global energy consumption. We thus evaluate the energy consumption of every commit `x` and we compare it to its `x-1` commit. We run every JMH benchmark for multiple iterations on a fixed amount of time (enough time to run the benchmark at least one), and we extract between 100 and $1,000$ energy measurements depending on the duration of each iteration. Thus, different commits can run a different amount of iterations within the time allowed to the JMH benchmark execution. This is why we consider the energy consumption of iterations rather than the whole benchmark, in order to have a correct estimation of the energy consumption for that commit. Then, we use the bootstrap method (Efron, 2000) to randomly build 100 subsets from the main set of measurements, and we compute the mean and standard deviation of these subsets. We end-up with 100 measures of averages and we use the median of these values for better accuracy and less bias.

The checked results are then used to build global statistics of the most efficient refactoring rules across the selected commits of all projects. We also pay special attention to the commits of each project that exhibit the most energy difference, when exceeding a threshold of 5%. This threshold is computed from the minimum CPU energy consumption variation and the computed standard deviation of the experiments (Ournani et al., 2020).

This additional check of those commits consists of applying a more detailed `git diff` analysis on the results of the previous step to understand every single occurrence of the detected refactorings and project the results and that there is no other changes that may affect the energy efficiency. Another check consists of an extra micro-benchmarking phase, where we prepare and execute the extracted refactorings to confirm and validate the effect they could have on the energy efficiency of the project/software. We also applied the Wilcoxon rank sum test (or Student test when possible) to check the statistical significance of the registered difference in the energy consumption between the commit `x` and the commit `x-1`, with a null hypothesis of the energy consumption of the commit `x` and `x-1` being equal with a 5% certainty. During our experiments, we were careful not to fall in the benchmarking crimes described in (van der Kouwe et al., 2018), so we can conduct robust and reproducible experiments and evaluations with a focus on energy consumption.

Most of our experimental setup is made available on GitHub, including all the used JMH Benchmarks, JSON extraction results, micro-benchmarks, CSV of measurements, scripts, etc.[3]

3 REFACTORING IMPACT ANALYSIS

In this section, we aim at answering our research questions with a clear conclusion on whether refactoring has a substantial impact on the evolution of

[3]https://anonymous.4open.science/r/c3d38dca-1ab2-4814-ba07-b182120c5739

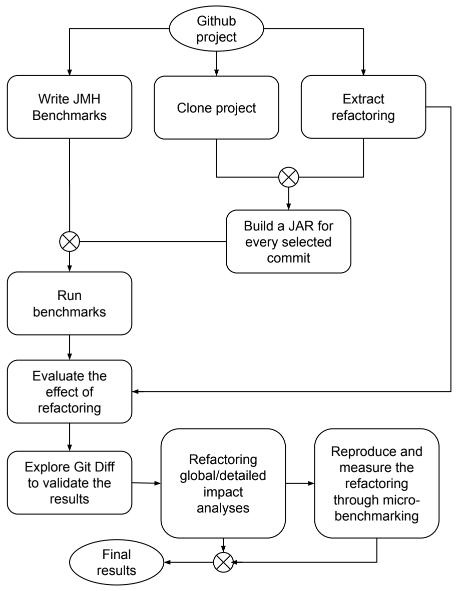

Github project
Write JMH Benchmarks
Clone project
Extract refactoring
Build a JAR for every selected commit
Run benchmarks
Evaluate the effect of refactoring
Explore Git Diff to validate the results
Refactoring global/detailed impact analyses
Reproduce and measure the refactoring through micro-benchmarking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

Figure 2: Methodology of refactoring analysis.

software energy consumption over time. We, therefore, conducted a set of experiments and validations to investigate the effect of structural refactoring on the evolution of software energy consumption.

3.1 Software Energy Consumption Evolution

The first step is to investigate the evolution of software energy consumption over time. Figure 3 depicts the evolution of energy consumption for the projects Google-Http, XStream, JGraphT, and Eclipse Collections, for which we run the main releases and report on the energy consumption measured over time, by focusing on the main functions stressed by our JMH benchmarks.

Except for JGraphT, one can observe that energy consumption tends to decrease over time for most of the projects. One can mention a 10% decrease in 12 months for the Google-Http project (cf. Figure 3a), a 10% decrease in 4 years for the Eclipse Collections project (cf. Figure 3c), and a very substantial decrease of 50% in 6 years for the XStream project (cf. Figure 3d).

Then, to have a more concrete look on the evolution of energy consumption per commit, we select the Gson project to reproduce the evolution of its energy consumption along the full commit history. Given the large number of involved commits, we consider the full set of commits of the Gson project (12 years) with a span of 25—*i.e.*, we build, run, and measure the energy consumption every 25^{th} commits. Figure 4 depicts the evolution of energy consumption for the Gson project with a total of 57 successfully reproduced commits, out of 60. The line plot validates and confirms the results shown in Figure 3.Most notably, one can observe a reduction of 82% from the highest to the lowest consumption commit within 12 years of the project's lifespan—*i.e.*, the energy consumption became 5 times lower. One can also see a more sudden energy consumption reduction between commits 600 and 900. This requires further investigation in the future.

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

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

3.2 Refactoring Rules Impact

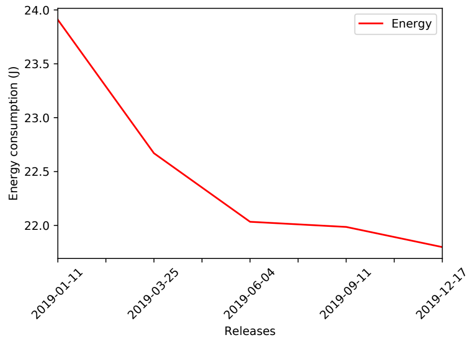
To dive into the effective impact that code refactoring may have on software energy consumption, we further tracked and analyzed the evolution of the energy consumption on commits where code refactorings were detected. Thus, in our study, we consider the full commit history of 7 open-source projects, and we analyze the impact on energy consumption of commits including code refactorings, as described in Section 2.

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

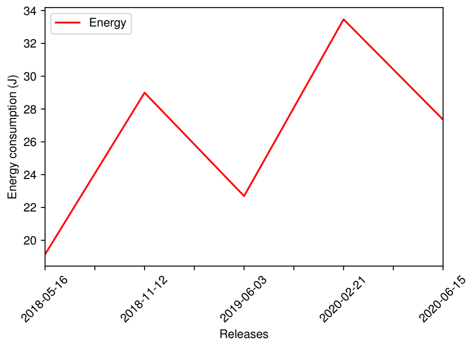
Then, we report on global statistics from the raw measurements we obtained from each project, thus establishing a summary of the most used code refactorings and their impact.

3.2.1 Global Code Refactoring Statistics

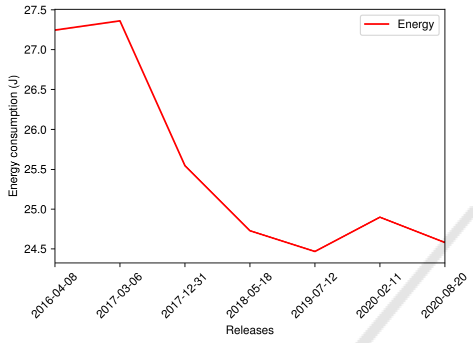
The purpose of this step is to highlight the most used/impactful code refactorings. While it is easy to identify the most used code refactorings by counting the number of occurrences of each refactoring rule and the commits they appear in, there is no consensus on how to measure the effective impact of code refactorings on energy consumption, if any. The large majority of commits comes with a set of code refac-



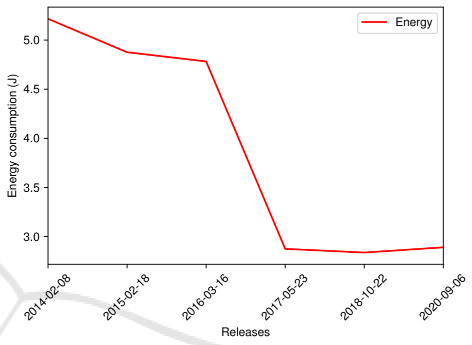
(a) Google-Http energy consumption over 11 months.



(b) JGraphT energy consumption over 2 years.



(c) Eclipse Collections energy consumption over 4 years.



(d) XStream energy consumption over 6 years.

Figure 3: Energy consumption evolution of Google-Http, XStream, JGrapht, and Eclipse Collections.

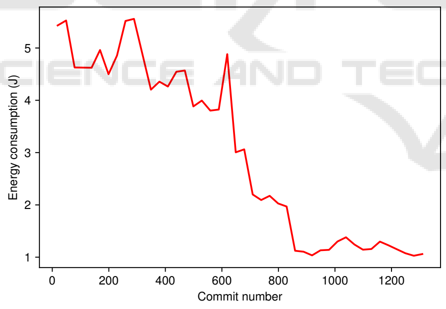


Figure 4: Gson energy consumption across for every 25th commit.

torings of many types, and even if these refactorings can impact the energy consumption, there is no trivial way to isolate such an impact for each type of refactoring. Thus, we consider 3 indicators to capture the energy impact of refactoring. The first indicator, *Impact in Commits* (IC), is the ratio between the number of commits where the refactoring had a positive or negative impact—*i.e.*, the commit x containing this refactoring consume more or less energy than the previous commit x-1—and the total number of commits containing this refactoring. Equation 1 therefore computes IC for a rule $r \in R$ by exploring all the commit history C of a given project:

$$\mathrm{IC}(r) = \frac{\sum_{c \in C} count\_positive\_negative(c, r)}{\sum_{c \in C} count(c, r)} \quad (1)$$

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

$$\mathrm{WIC}(r) = \frac{\sum_{c \in C} count\_positive\_negative(c, w_r)}{\sum_{c \in C} count(c, w_r)} \quad (2)$$

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

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

$$\delta\%(r) = \frac{\sum_{x=1}^{C_r}(E_x - E_{x-1})}{\|C_r\|} \quad (3)$$

$$\delta|\%|(r) = \frac{\sum_{x=1}^{C_r}|E_x - E_{x-1}|}{\|C_r\|} \quad (4)$$

where E_x and E_{x-1} represent the mean energy consumption of the commit x that includes at least the refactoring r, and the energy consumption of the commit x-1, respectively. These indicators are complementary to reflect the impact of the code refactorings on the energy consumption. Therefore, we consider an aggregate indicator that combines the previous indicators to capture the energy impact of refactorings across commits. This indicator, named *Refactoring Impact* (RI) builds on the previous indicators: the higher WIC and $\delta|\%|$, the most impactful the refactoring r is. However, if the difference $\delta|\%| - \delta\%$ is high, it means that the refactoring r has an unpredictable effect on the energy consumption and may affect the energy consumption positively or negatively. This is a negative effect and could mean that the refactoring does not have any impact at all. On the other hand, the more commits we have with the refactoring r, the more certain we are of the effect that it could have. Thus, we use the exponential function in Equation 5 so the denominator cannot be null.

$$\mathrm{RI}(r) = \frac{\mathrm{WIC}(r) \times \delta|\%|(r)}{e^{\delta|\%|(r) - \delta\%(r)}} \times \|C_r\| \quad (5)$$

Table 2 shows the computed indicators for a total of 25 mined refactoring rules. We note that the commits that could not be reproduced and those where the refactorings are parts of classes that are not tested by our benchmark have already been discarded and not displayed in Table 2. Before analyzing the results we excluded all the code refactorings with a low number of occurrences and/or commits (less than 20 CountxCommits). For example, code refactorings that occurred only a couple of times and/or only in one or two commits cannot be faithfully studied due to insufficient data. Then, we highlight (in Cyan) the refactoring rules that have the best values for the previous indicators, which are very likely the refactorings with the most impact on energy consumption. The 4 refactoring rules with the most number of occurrences and commits, with a minimal IC of 30%, are "add method annotation", "rename parameter", "add class annotation", and "move class". These refactoring rules are also those that exhibit the highest RI, and thus, are most likely to be the most impactful on energy consumption. However, we still have to assess that these refactoring rules have an effective impact on the evolution of energy consumption. Thus, we conducted a more detailed study on the commits with the highest impact to validate the effect of code refactorings on energy consumption.

3.2.2 Diving into the Most Impactful Commits

With the most impactful commits, we refer to commits where we observed the most substantial energy differences between the commits x and commit x-1. To select these commits, we fix a threshold of 5% in energy consumption difference. This threshold was fixed based on the CPU energy consumption variation (Ournani et al., 2020) and the standard deviation of the many executions we ran on the same test, which is often around 4% to 5%. A total of 7 commits have been retrieved from the projects Gson, JFlex, Eclipse-Collections and JGraphT (no other refactoring commit with a minimal impact of 5% has been observed among the other projects). We note that our experimental setup would highlight any effect that these refactoring could have caused on energy consumption. Indeed, the execution of a JMH code, which uses the compiled JAR for the commit x, is composed of numerous warmup and standard iterations. Each iteration itself consists of running the benchmark many thousands of times for several seconds, so the effect that difference between the commits x and x-1 could be noticed, if any.

Table 3 reports on the most impactful commits including code refactorings. For each commit, we can see the type and number of refactorings extracted using REFACTORINGMINER (Tsantalis et al., 2018; Tsantalis et al., 2020), the measured energy consumption difference, a short description of the refactoring-related changes that have been observed within the commits, and the computed *p*-value of the Wilcoxon test.

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

Table 2: The observed impact of mined refactoring rules.

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

consumption, with add annotation and move class being the most common. Sometimes, they are the only detected code refactorings, that we could suspect to be responsible for the energy consumption variation, as in commit `#b9dfbc` of Eclipse Collections.

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

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

The remaining commits—being `#827717`, `#45bf2d`, and `#298b7a`—mainly contain the add annotation and move class refactorings. We thus achieve our third validation step through dedicated micro-benchmarking. We first build a micro-benchmark to check the effect that every encountered annotation may have(`@override`,

Table 3: A deeper look into the most impactful commits.

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

`@SuppressWarnings("unchecked")`, `@SuppressWarnings("unused")`) and ran hundreds of millions times each, on classes, methods and variables to check whether it has an effect on the energy consumption. The results—as expected—did not have any effect (about 1% difference that we cannot consider due to CPU energy variations (Ournani et al., 2020)) on energy consumption, as annotations are not supposed to have a substantial impact on the generated bytecode that would be executed by the JVM. This would invalidate the fact that the observed energy consumption difference is mainly related to the add annotation refactoring in the commits that only contain this type of refactoring, such as `#827717`, `#b9dfbc`, and `#298b7a`. The second micro-benchmark concerns the `move class` refactoring, where we measured the energy consumption for several scenarios, after moving some classes/interfaces around and reorganizing the structure of the micro-benchmark. The results showed a difference in energy consumption of up to 8%, with an average standard deviation of 5%. The `move class` refactoring—which is often accompanied with the rename refactorings—indicates a code reorganization that might have an impact. While the observed impact through the JMH experiments or with micro-benchmarking might not be substantial, it would be beneficial to be aware that restructuring/reorganizing a project could have an impact on energy consumption, and thus compare the before/after energy consumptions to track that effect. Unfortunately, we could not detect any specific pattern or guidelines on when the code reorganization or restructuring would impact positively or negatively the energy consumption. Hence, we can only faithfully retain the commit `#45bf2d` of the Gson project among the commits of Table 3, where the 30 move class refactoring could have been responsible of 2% of energy consumption difference as the standard deviation of the measures is 5%.

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

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

4 RELATED WORK

In this section, we review the state of the art of green software design efforts related to code refactorings.

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

In a more detailed study of the impact of only one refactoring rule "*inline method*" on 3 Java applications, (W G P Silva et al., 2010) reported that the impact on the execution time and energy consumption that was expected to be positive, was not always true.

Rather than looking for green refactoring rules reducing software energy consumption, some practitioners chose to conduct wider studies that apply on a much larger set of refactorings to capture a subset of "green" rules. This is exactly what the authors of (Jae-Jin Park et al., 2014) pursued: They prepared C++ micro-benchmarks of 63 refactoring techniques/design patterns suggested by Martin Fowler (10., 1999), then ran experiments and isolated a set of green refactoring rules based on the micro-benchmarks for C++.

The authors of (Kumar et al., 2017) focused on investigating the impact of Java coding practices, which include primitive data types, operations on strings, usage of exceptions, loops, and arrays. Using RAPL (Khan et al., 2018), they measure the energy consumption of code snippets and micro-benchmarks and presented some minor observations, such as string concatenation consuming less than `StringBuilder` and `StringBuffer`, static variables consume 60% more energy compared to instance variables, etc.

Mobile Applications. In another context, the reduction of software energy consumption through refactoring actions has also been explored in the context of mobile applications. EARMO proposes a multi-objective refactoring approach to automatically improve the architecture of mobile applications (Morales et al., 2018). The authors conducted an empirical study to measure the negative impact of 8 anti-patterns on 20 open-source applications. They then used a multi-objective search-based approach, called EARMO, to correct up-to 84% of the anti-patterns on the tested applications and increase the battery lifespan by up-to 29 minutes. However, their statistical analyses with a significance level of 5% only showed that half of the rules can impact energy efficiency in some cases. Moreover, the CPU/chip energy variation has not been taken into account for the significance level.

Other works also considered energy efficient refactoring for mobile applications (Gottschalk et al., 2013). In particular, the authors of (Rodriguez, 2017) presented some early experiments on different micro-benchmarks and discussed many coding aspects with a focus on implementation techniques, such as how to iterate on a matrix, avoid operations with immutable data types, evaluating strings, or the use the more specific numeric data types to save battery life. Anwar *et al.* (Anwar et al., 2019) also gave concrete examples on how to save some battery time through refactoring. They achieved a maximum of 10% of energy savings by refactoring the DuplicatedCode and TypeChecking code smells.

Cruz *et al.* (Cruz and Abreu, 2017) studied the effect of 8 of the best performance-based practices on the energy efficiency of 6 Android applications. The results of the experiments showed that some patterns, such as ViewHolder, DrawAllocation, WakeLock, ObseleteLayoutParam need to be taken into account for a better design of energy-efficient applications, with a reported impact of 4.5% for the Writeily Pro app. The authors also proposed the LEAFACTOR tool to improve the energy efficiency of Android applications by automatically refactoring the source code to fix the above patterns (Cruz et al., 2017). The process was applied on a set of 140 open-source Android applications and yielded a total of 222 refactorings, which were submitted as pull requests, with 16 successfully merged pull requests. Unfortunately, the paper does not discuss any further energy enhancements of the applied LEAFACTOR refactorings on the original applications. While the reported impact is still relatively small, most of the covered patterns are related to screen/sensors usage that are very specific to mobile applications and cannot be generalized.

Beyond the State of the Art. While energy variation and measurement errors inherently represent a serious threat to the accuracy of the previous works considered in this paper, our results do not contradict observations reported in the context of mobile applications. Indeed, our study focuses on assessing the energetic impact of structure-oriented code refactorings that have been mined from 7 long-existing Java desktop projects. While the comparison with other works that focused os a assessing the energetic impact of a set of code refactoring rules on different scenarios may be not completely fair due to the eventual differences of the considered refactoring rules and type of applications, we still noticed that the registered impact for server/desktop applications of structure-oriented refactorings is usually less than 5% (Jae-Jin Park et al., 2014) (Moreira et al., 2020) and sometimes not even stable (W G P Silva et al., 2010). This is highly related to the energy variation (Ournani et al., 2020). Otherwise, code refactorings can have a different impact on mobile applications as discussed by Palomba *et al.* (Palomba et al., 2019), Linares-Vasquez *et al.* (Linares-Vásquez et al., 2018) Iannone *et al.* (Iannone et al., 2020) and Agolli *et al.* (Agolli et al., 2017), as they can directly impact alternative hardware resources, such as the device display and sensors, with a more effective impact on energy consumption. Yet, we propose an empirical methodology to analyze the impact of refactorings on the energy consumption of any software. We thus believe that our methodology would deserve to be considered in the context of server applications in order to assess more specific code refactorings applied along the lifespan of such software systems.

5 THREATS TO VALIDITY

There are a couple of issues that can impact the accuracy of our results. First, our analysis highly depends on the REFACTORINGMINER tool and its ability to extract every single occurrence of each of the 55 refactorings it supports. Moreover, there are some other refactorings, not listed among the 55, that have not been extracted and thus considered in our study, especially those related to implementation and computation details and those that cannot be discovered automatically. During our experiments, we use Intel RAPL to measure energy consumption. It is one of most accurate tools to measure CPU and DRAM energy consumption (Desrochers et al., 2016), but only reports on the global energy consumption, which includes the OS and the other processes running with the software system under study. We thus conducted experiments on a minimal OS setup and disabled all optional daemons and services, along with other guidelines and best practices in order to reduce the error margin and the CPU energy variation to the minimum (Ournani et al., 2020).

We also focused on running benchmarks that last for many seconds (around 150 sec for Gson, 450 sec for XStream, 330 sec for OkHttp, 290 sec for Google-Http, 780 sec for JGraphT, 720 sec for JFlex, and 600 sec for Eclipse Collections), so we can obtain trustful and robust evaluations of the potential impact of changes between commits with an overall continuous execution time of experiments that exceeded 100 hours.

The manual steps in our study remain the design of the JMH benchmarks and some checks of the git diffs. In the first case, we tried to write benchmarks that stress the main purpose or functionality of each project, so we can ensure that the comparison is based on the same functionalities that are available on all commits and versions. While this is moderately easy for some projects, such as Gson or XStream, it is much more complicated for other projects, such as

Eclipse Collections where many collections and operations are available and can change. We tried in this case to cover many functionalities that are available in most commits, even if it requires some adjustments and adaptation when projects are restructured / reorganized between versions. Regarding git diff, we gave the major importance to the commits with the most impact, as it is not possible to meticulously check all the changes on all the selected commits. Another threat may be related to our selection of the commits with the most refactoring to have a reasonable execution time. Even if selected commits are most likely to be the most impactful. It results in a low number of selected commits among the global set of commits.

6 CONCLUSION

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

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

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

REFERENCES

(1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA.

Abid, C., Alizadeh, V., Kessentini, M., do Nascimento Ferreira, T., and Dig, D. (2020). 30 years of software refactoring research: A systematic literature review. *CoRR*, abs/2007.02194.

Agolli, T., Pollock, L. L., and Clause, J. (2017). Investigating decreasing energy usage in mobile apps via indistinguishable color changes. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 30–34. IEEE.

Anwar, H., Pfahl, D., and Srirama, S. N. (2019). Evaluating the Impact of Code Smell Refactoring on the Energy Consumption of Android Applications. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 82–86, Kallithea-Chalkidiki, Greece. IEEE.

Bree, D. C. and Cinnéide, M. Ó. (2020). Inheritance versus delegation: which is more energy efficient? In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*, pages 323–329. ACM.

Chowdhury, S. A., Hindle, A., Kazman, R., Shuto, T., Matsui, K., and Kamei, Y. (2019). Greenbundle: an empirical study on the energy impact of bundled processing. In Atlee, J. M., Bultan, T., and Whittle, J., editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1107–1118. IEEE / ACM.

Cruz, L. and Abreu, R. (2017). Performance-based guidelines for energy efficient mobile applications. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 46–57. IEEE.

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

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

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

Fonseca, A., Kazman, R., and Lago, P. (2019). A manifesto for energy-aware software. *IEEE Softw.*, 36(6):79–82.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Gottschalk, M., Jelschen, J., and Winter, A. (2013). Energy-Efficient Code by Refactoring. *Softwaretechnik-Trends*, 33(2):23–24.

Iannone, E., Pecorelli, F., Nucci, D. D., Palomba, F., and Lucia, A. D. (2020). Refactoring android-specific energy smells: A plugin for android studio. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 451–455. ACM.

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

Kerievsky, J. (2004). *Refactoring to Patterns*. Pearson Higher Education.

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

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

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D. (2014). Mining energy-greedy API usage patterns in Android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 2–11, Hyderabad, India. ACM Press.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Penta, M. D., Oliveto, R., and Poshyvanyk, D. (2018). Multi-objective optimization of energy consumption of guis in android apps. *ACM Trans. Softw. Eng. Methodol.*, 27(3).

Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., and Clause, J. (2016). An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, pages 237–248, Austin, Texas. ACM Press.

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

Morales, R., Saborido, R., Khomh, F., Chicano, F., and Antoniol, G. (2018). EARMO: an energy-aware refactoring approach for mobile apps. *IEEE Trans. Software Eng.*, 44(12):1176–1206.

Moreira, E., Correia, F. F., and Bispo, J. (2020). Overviewing the liveness of refactoring for energy efficiency. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, pages 211–212, Porto Portugal. ACM.

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

Palomba, F., Nucci, D. D., Panichella, A., Zaidman, A., and Lucia, A. D. (2019). On the impact of code smells on the energy consumption of mobile applications. *Inf. Softw. Technol.*, 105:43–55.

Pinto, G., Castor, F., and Liu, Y. D. (2014). Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '14*, pages 345–360, Portland, Oregon, USA. ACM Press.

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

Pinto, G., Soares-Neto, F., and Castor, F. (2015). Refactoring for Energy Efficiency: A Reflection on the State of the Art. In *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*, pages 29–35, Florence, Italy. IEEE.

Rodriguez, A. (2017). Reducing Energy Consumption of Resource-Intensive Scientific Mobile Applications via Code Refactoring. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 475–476, Buenos Aires, Argentina. IEEE.

Rodriguez-Cancio, M., Combemale, B., and Baudry, B. (2016). Automatic microbenchmark generation to prevent dead code elimination and constant folding. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 132–143, New York, NY, USA. Association for Computing Machinery.

Samir Hasan, Rachary King, and Munawar Hafiz (2016). Energy Profiles of Java Collections Classes. In *ICSE*.

Tsantalis, N., Ketkar, A., and Dig, D. (2020). Refactoringminer 2.0. *IEEE Transactions on Software Engineering*.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA. ACM.

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

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

W G P Silva, Lisane Brisolara, Corrêa, U. B., and Carro, L. (2010). Evaluation of the impact of code refactoring on embedded software efficiency. *Unpublished*.