How Software Refactoring Impacts Execution Time

LUCA TRAINI, DANIELE DI POMPEO, and MICHELE TUCCI, University of L'Aquila, Italy
BIN LIN, Software Institute - USI, Lugano, Swizerland
SIMONE SCALABRINO, University of Molise, Italy
GABRIELE BAVOTA and MICHELE LANZA, Software Institute - USI, Lugano, Swizerland
ROCCO OLIVETO, University of Molise, Italy
VITTORIO CORTELLESSA, University of L'Aquila, Italy

Refactoring aims at improving the maintainability of source code without modifying its external behavior. Previous works proposed approaches to recommend refactoring solutions to software developers. The generation of the recommended solutions is guided by metrics acting as proxy for maintainability (e.g., number of code smells removed by the recommended solution). These approaches ignore the impact of the recommended refactorings on other non-functional requirements, such as performance, energy consumption, and so forth. Little is known about the impact of refactoring operations on non-functional requirements other than maintainability.

We aim to fill this gap by presenting the largest study to date to investigate the impact of refactoring on software performance, in terms of execution time. We mined the change history of 20 systems that defined performance benchmarks in their repositories, with the goal of identifying commits in which developers implemented refactoring operations impacting code components that are exercised by the performance benchmarks. Through a quantitative and qualitative analysis, we show that refactoring operations can significantly impact the execution time. Indeed, none of the investigated refactoring types can be considered "safe" in ensuring no performance regression. Refactoring types aimed at decomposing complex code entities (e.g., Extract Class/Interface, Extract Method) have higher chances of triggering performance degradation, suggesting their careful consideration when refactoring performance-critical code.

CCS Concepts: • **Software and its engineering → Software performance**; **Empirical software validation**; **Maintaining software**; **Software evolution;**

Additional Key Words and Phrases: Software maintainability, performance, execution time, refactoring

Lin, Bavota, and Lanza are grateful for the financial support by the Swiss National Science foundation through SNF Project SENSOR (183587).
Di Pompeo is grateful for the financial support by CIPE-RESTART funding through Ex-EMERGE project.
Authors' addresses: L. Traini, D. Di Pompeo, M. Tucci, and V. Cortellessa, University of L'Aquila, L'Aquila, Italy; emails: luca.traini@univaq.it, {daniele.dipompeo, michele.tucci, vittorio.cortellessa}@univaq.it; B. Lin, G. Bavota, and M. Lanza, Software Institute - USI, Lugano, Lugano, Swizerland; emails: {bin.lin, gabriele.bavota, michele.lanza}@usi.ch; S. Scalabrino and R. Oliveto, University of Molise, Pesche (IS), Italy; emails: {simone.scalabrino, rocco.oliveto}@unimol.it.

1049-331X/2021/12-ART25 $15.00
https://doi.org/10.1145/3485136

ACM Reference format:
Luca Traini, Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. 2021. How Software Refactoring Impacts Execution Time. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 25 (December 2021), 23 pages.
https://doi.org/10.1145/3485136

1 INTRODUCTION

Software systems are continuously changed to meet new requirements, fix defects, and enhance existing features. A key point for sustainable software evolution is high-quality source code. Indeed, several empirical studies have provided evidence that low code quality hinders maintenance and evolution activities [21, 30]. Tools have been developed to recommend to developers how to improve code quality via refactoring operations (i.e., refactoring recommender systems) [3].

Despite their benefits, most tools ignore the heterogeneity of modern software and the different priorities that non-functional requirements (e.g., maintainability, performance, security) may have in different contexts. For example, smartphones have limited battery life and require software optimized to reduce energy consumption, while embedded systems often come with performance-critical requirements specifying precise time windows in which a task must be executed.

State-of-the-art refactoring recommenders target the improvement of code quality from a narrow perspective, focusing on improving code readability or removing well-known anti-patterns or code smells [4, 28, 45]. Basically, they *aim at improving code maintainability without considering the possible side effects that the recommended refactorings may have on other, maybe more important, non-functional requirements*. In other words, they do not consider the priority that different non-functional requirements may have. For this reason, some researchers started investigating the impact of "maintainability-driven" refactorings on other non-functional attributes.

Sahin et al. [35] showed that refactoring can change the amount of energy used by a software application, while Demeyer [11] investigated the impact on performance of introducing virtual functions in C++ code. These studies started laying the empirical foundations for building more *sensible* refactoring recommender systems, able to consider trade-offs between multiple non-functional requirements when making recommendations. The only concrete example is the EARMO tool by Morales et al. [29], able to support the refactoring of mobile apps by removing anti-patterns while controlling for the energy efficiency of the app. There is still a lack of empirical knowledge about the impact of refactoring on non-functional requirements.

We present a comprehensive study to investigate the impact of 16 different types of refactoring on the execution time of 20 Java systems. The systems have been selected given their attention to execution time, demonstrated by the presence of performance benchmarks in their code repositories. Using RefactoringMiner [46], we mined the subject systems for "refactoring commits," i.e., commits that contain refactoring operations. Each refactoring commit is accompanied by the code components (in our study, methods or classes) impacted by the refactoring. We manually inspected each commit to ensure that refactoring was its only goal. Through dynamic code analysis, we identified the code components executed by the performance benchmarks in each system and the refactoring operations that impacted them. Overall, we collected 82 commits implementing 167 refactoring operations impacting performance-relevant components. Each commit provides several data points for our study, since refactorings implemented in the same commit can impact different performance-relevant components and, thus, exercise different performance benchmarks. The total number of data points involved in our study (i.e., pairs of refactoring actions, benchmarks) is 1,598. The collection of this data required ~476 machine days. Besides presenting quantitative results showing the impact of (different types of) refactoring on execution time, we also qualitatively

analyze cases in which refactoring had a negative impact on execution time, distilling lessons learned useful to (1) developers, for avoiding specific refactoring scenarios when performance is key, and (2) researchers, for developing performance-aware refactoring recommenders.

Our results show that refactoring can have a substantial impact on execution time, in both a positive and negative way. About 55% of the "refactoring commits" cause a statistically significant performance change in at least one performance benchmark. Moreover, certain types of refactorings are more prone to degrade execution time and should be carefully performed in performance-critical systems. For example, Extract Class and Extract Method induce regressions, respectively, in 16% and 12% of impacted performance benchmarks.

2 STUDY DESIGN

The *goal* of the study is to investigate the impact of refactoring operations on software performance. Measuring performance encompasses multiple metrics, such as response time, utilization, and so forth. In the context of this article, we focus on execution time, intended as the time that a section of code needs to be executed, without any concurrency or resource sharing with other software running on the same platform.

The *context* is represented by 82 commits mined from 20 systems in which developers performed a total of 167 refactoring operations impacting code components exercised by performance benchmarks. The study answers the following **research questions (RQs)**:

- RQ_1 *To what extent do developers refactor performance-relevant code components?* We want to understand if developers are reluctant to refactor performance-relevant parts of the system.
- RQ_2 *What is the impact of refactoring on performance?* We analyze the relationship between refactoring and performance, computing the percentage of cases in which refactoring improved, deteriorated, or did not impact performance.
- RQ_3 *What types of refactoring operations are more likely to impact performance?* We investigate the relationship between types of refactoring operations (e.g., Extract Method) and performance. Besides quantitatively analyzing our findings, we report interesting examples in which the refactoring had a negative impact on performance and we distill lessons learned useful for both researchers and practitioners.

2.1 Data Collection

We describe the procedure we followed to collect the data needed for our RQs. Specifically, we (1) selected Java open-source projects with performance benchmark suites, (2) detected refactoring operations to assess their performance impact, and (3) ran benchmarks before and after the refactoring operations were performed. We report in Figure 1 an overview of the process we used to collect our data.

2.1.1 Project Selection. We selected projects in which developers defined micro-benchmarks for performance assessment. To do so, we queried GitHub for Java projects having a dependency with Java Microbenchmarking Harness (JMH),[1] the de facto standard for micro-benchmarks. While other micro-benchmarking tools are available (e.g., Caliper, Japex, or JUnitPerf), they are either less popular than JMH, discontinued, or not executable in an automated way [25, 40].

We used the GitHub APIs to obtain the list of the 1,000 most recently indexed projects that (1) used Maven as the dependency manager (i.e., they had at least a file named `pom.xml`) and (2) had an explicit dependency with `org.openjdk.jmh.jmh-core`, i.e., the core library required to run JMH. We considered only projects having at least 100 stars, and 88 projects satisfied this

[1] JMH, https://openjdk.java.net/projects/code-tools/jmh/.

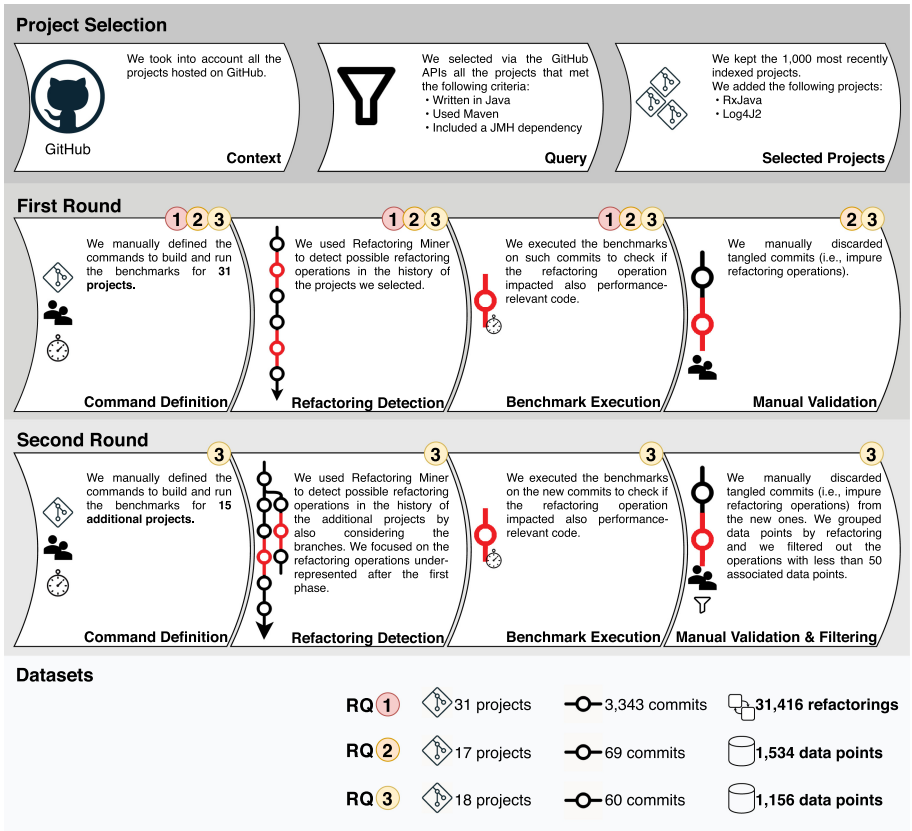

Fig. 1. Overview of our data collection process. We report, on top of each step, the RQs for which it was done. We report below summary information about the three datasets used to answer our research questions.

criterion. In addition to them, we considered two popular Java projects already used in a previous microbenchmark-related study [24], i.e., RxJava and Log4j2. They were not included in the set of projects we initially selected because they were not among the 1,000 most recently indexed. Additionally, RxJava uses Gradle instead of Maven as a build tool.

We manually analyzed the list of projects to find the commands that would build and run the benchmark suite. In most of the cases build commands are of the form `mvn -pl [jmhModule] -am package`, where `[jmhModule]` is the Maven module[2] dedicated to performance benchmarks. These commands usually build a jar file `[benchmarks.jar]` that contains performance benchmark suites along with all their dependencies (e.g., system code). Benchmark suites can then be executed through a run command, such as `java -jar [benchmarks.jar]`. To identify build and run commands, we analyzed the GitHub pages of projects. In the simpler cases, commands are explicitly reported in README files (e.g., `JCTools/JCTools`[3] and `cantaloupe-project/cantaloupe`[4]). In other cases, we derived them through a manual analysis

[2]Maven modules, https://bit.ly/3con0VI.
[3]JCTools Benchmarks, https://bit.ly/3r6rmVC.
[4]Cantaloupe, https://bit.ly/3lFllyf.

of `pom.xml` files (e.g., `apache/logging-log4j2`[5] and `eclipse/jetty.project`[6]). We were able to identify working commands and runnable benchmarks for 31 projects (including RxJava and Log4j2).

2.1.2 Refactorings and Benchmarks Gathering. We used RefactoringMiner [46] to extract refactoring operations performed on the default branch of each project. RefactoringMiner is able to detect 55 types of refactoring operations (e.g., Extract Method, Extract Class, etc.). Given the time-consuming nature of our data collection, we discarded refactoring operations likely to have negligible or no performance impact. Specifically, we did not consider seven refactoring types: *Rename-related* refactoring operations (Rename Method, Rename Class, Rename Variable, Rename Parameter, and Rename Attribute) and *package-related* refactoring operations (Change Package and Move Class).

Given a git repository of a Java project, RefactoringMiner reports (1) the commits featuring refactorings, (2) the refactoring types, and (3) the files and lines affected by the refactoring. We collected 494,826 refactoring operations (of 48 different types) performed in 43,008 commits and 31 projects. We identified benchmarks suitable to evaluate their performance impact by verifying for each benchmark b in the project whether the code affected by refactoring operations is exercised by b. We first derived, for each refactoring r performed within the commit c, the set of Java methods impacted by r, namely M_r^c. We used srcML [8] to parse the Java files affected by r and we identified the set of methods M_r^c based on the impacted lines of code returned by RefactoringMiner.

For each benchmark b and for each commit c returned by RefactoringMiner, we derived the set M_b^c of Java methods executed by b. We built system snapshots both before and after the commit c is performed and ran dynamic analysis on the benchmarks to identify methods invoked by them. If we were not able to build one of the two snapshots or to run the benchmark suite, we discarded the commit c. We discarded 39,665 commits, while 3,343 are retained. The large number of discarded commits can be explained by two main reasons. First, performance benchmark suites are usually introduced at later stages of systems history. For example, we found that 22,938 out of the 39,665 discarded commits (~58%) are removed because the project did not have yet a benchmark suite. Second, several commits correspond to "unstable" states of the system, which makes it unfeasible to build the related snapshots and/or run benchmarks. We ran each benchmark b for 1 second and recorded the methods invoked in the execution using Java Flight Recorder (JFR)[7] to derive M_b^c. This required ~221 machine days, involving the profiled execution of more than 4M benchmarks across 7,901 systems snapshots.

Finally, we intersected the list of methods affected by refactoring operations with those executed by benchmarks to identify benchmarks suitable to assess the performance impact of refactoring operations. We identified a set of 3,533 data points. Each data point is denoted by a tuple (p, c, b, R), where p indicates the project, c the commit, b one of the project's benchmarks, and R a subset of refactoring operations performed in c. Each data point is such that every refactoring $r \in R$ modifies at least one method executed by b, i.e., $M_r^c \cap M_b^c \neq \emptyset$. Note that a refactoring operation may impact multiple locations in the code. For example, an Extract Method refactoring typically involves a source method and a target method. In our study, a benchmark b is considered suitable to evaluate the impact of a refactoring r if it executes at least one of the methods that are impacted by r. The 3,533 data points involved 201 commits, 521 refactoring operations (of 24 types), and 26 projects.

It is worth noting that a commit c may be a tangled commit [17], involving other code changes (apart from refactorings) that can affect parts of code executed by b. Including tangled commits

[5] Apache Log4j 2, log4j-perf pom file. https://bit.ly/3r7zSDN.
[6] Eclipse Jetty, jetty-jmh pom file. https://bit.ly/3lC2vYV.
[7] JFR, https://tinyurl.com/y7yq8xc2.

might mislead the assessment of the impact on performance of refactoring operations. Therefore, we filtered out such cases by performing manual analysis on all the data points to verify whether refactoring operations are the only code changes in the commit that affect the code executed by b.

We grouped the data points by commit and randomly assigned them to five of the authors, who manually analyzed the data points (p, c, b, R) assigned to them by inspecting the diff of the commit c, the commit message accompanying c, the set of methods invoked by the benchmark b (namely M_b^c), and the discussions in the issue tracker related to c (if a link to the discussions could be identified). Specifically, each author inspected the diff of any commit c on the GitHub website with the help of a Google Chrome extension named *Refactoring Aware Commit Review*. Such extension uses RefactoringMiner to visually augment the diff with refactoring information. It is able to highlight the type of refactoring, where it occurred in the diff, and which parts of the diff are identified as "Added code" (new code), "Same code" (code that was only moved), and "Method call" (call to existing code that was moved). The goal of the manual analysis was to decide whether b can be reliably used to evaluate the performance impact of the refactoring operations R (i.e., no other interfering changes were impacting the methods exercised by b besides the refactoring operations). If an author classified a data point as relevant for our study (i.e., a pure refactoring impacting b), it was double-checked by another author. We only kept data points confirmed as relevant for our study by two of the authors.

The dataset resulting from this process contains 1,534 data points involving 69 commits, 150 refactoring operations, and 16 refactoring types across 17 projects.

2.1.3 Benchmarks Execution/Performance Data Collection. The performance comparison of different software versions in Java applications is far from trivial. There are a number of sources of non-determinism, such as Just-In-Time (JIT) compilation and optimizations in the Java Virtual Machine (JVM) [13]. We relied on steady-state performance [13]: we repeated a benchmark execution for several *iterations* and collected measurements only after a steady state had been reached. Indeed, first iterations (often called *warm-up iterations*) are subject to noise due to performance variations in transient states, usually caused by class loading and JIT (re)-compilation. Hence, in our experimental setup, measurements are only collected in iterations that are subsequent to *warm-up*, namely *measurement iterations*. Different VM invocations running multiple benchmark iterations may result in different steady-state performance data. For this reason, we also repeated benchmark iterations multiple times on different VM invocations.

JMH allows to define the number of warm-up iterations, measurement iterations, and VM invocations directly in Java code or via command line arguments. We used the number of iterations defined in the code by benchmark developers for warm-up and measurement iterations, and we fixed the number of VM invocation to 10 (the JMH default) as done in previous studies [13, 23].

Given the previously described dataset, for each data point (p, c, b, R), we built system snapshots for the project p both before and after the commit c was performed, and we executed b on both snapshots, collecting two sets of measurements: E^{before} and E^{after}. Both of them are matrices, where $E_{i,j}$ represents the observed execution time for the jth benchmark iteration on the ith VM invocation. Execution of benchmarks for the 1,598 data points of our study required 79 machine days on a dedicated machine.

Although all data points of our dataset are suitable to evaluate the performance impact of refactoring operations in general (see RQ_2), many of them (~19%) cannot be used to analyze the relationship between types of refactoring operations and performance (RQ_3), since they involve multiple types of refactoring operations. We performed a second round of data collection specifically targeting the identification of commits in which a single type of refactoring was performed, focusing on data points (p, c, b, R) where all the refactoring operations $r \in R$ are of the same type.

To speed up the process, we removed 41 refactoring types from our data collection since, during the data collection performed for RQ_2, they had few related data points (<50). Also, we did not collect additional data points concerning Extract Method since we already had sufficient data points in our dataset (166 data points, 40 refactoring operations, 18 commits, and nine projects). As a result, the second round of data collection focused on six types of refactoring: Extract Superclass, Inline Variable, Extract Class, Move Method, Inline Method, and Extract Interface.

For the supplementary data collection we mined refactorings (of the six targeted types) by launching RefactoringMiner on all the branches of all 31 projects. We also derived commands to run and build benchmarks for 15 additional projects gathered from [24]. Then, we derived (p, c, b, R) data points, as described in Section 2.1.2, but discarded those having multiple types of refactoring operations in R. Finally, performance measurements were collected for each data point as described in Section 2.1.3. Overall, we found 64 additional data points, which involve 17 refactoring operations, spanning 13 commits and six projects. The profiled execution of benchmarks to derive M_b^c sets lasted 176 machine days. The execution of the benchmarks took ~11 machine hours.

This additional dataset is only used to analyze the relationship between types of refactoring operations and performance (RQ_3), while it is used neither to evaluate the performance impact of refactoring operations at coarse-grained level (RQ_2) nor to evaluate the density of refactoring operations in parts of the system known to be performance relevant (RQ_1). The rationale behind this decision is that the additional data points collected for RQ_3 are by construction only related to six refactoring types and ignore other refactoring operations performed by developers in the change history of the mined projects. When answering RQ_1 and RQ_2 it is important to use a dataset that reflects the actual distribution of refactoring types in the versioning system of the subject projects, something that would not happen by including the additional data points collected for RQ_3.

We collected 1,598 data points, 167 refactoring operations of 16 types, 82 commits, and over 20 projects.

2.2 Data Analysis

Answering RQ_2 and RQ_3 requires to determine whether refactoring operations have an effect (either positive or negative) on software performance. For this reason, we first describe the process we used to determine, for a given data point (p, c, b, R), whether refactoring operations R cause *regression*, *improvement,* or *unchanged performance* in benchmark b.

2.2.1 Reliably Detecting Performance Change. To determine whether refactoring operations lead to non-negligible performance change, we used the approach proposed by Kalibera and Jones to build confidence intervals for the ratio of mean execution times [19, 20]. Compared to other performance change detection techniques (e.g., hypothesis testing with Wilcoxon rank-sum combined with effect sizes [13, 23] and change detection through testing for overlapping confidence intervals [13, 23]), the main benefit of the Kalibera and Jones technique is that, in addition to a reliable performance-change detection, it provides a clear and rigorous account of the performance change magnitude and of the uncertainty involved. For example, it can indicate that a system version is slower (or faster) than another by $X\% \pm Y\%$ with 95% confidence. To build the confidence interval we used bootstrapping [10], with hierarchical random re-sampling [34] and replacement. Re-sampling was applied on two levels [20]: VM invocations and iterations.

We ran 1,000 bootstrap iterations. At each iteration, new realizations of E^{before} and E^{after} measurements were simulated and the relative performance change was computed. The simulation of the $\hat{E}^{before}$ new realizations randomly selected a subset of real data from E^{before} with replacement.

Similarly, $\hat{E}^{after}$ was simulated by randomly sampling E^{after}. The two means (μ^{before} and μ^{after}) and the relative performance change (ρ) for simulated measurements were computed as follows:

$$\mu^{before} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}\hat{E}_{i,j}^{before}}{mn} \quad \text{and} \quad \mu^{after} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}\hat{E}_{i,j}^{after}}{mn} \quad \text{and} \quad \rho = \frac{\mu^{after} - \mu^{before}}{\mu^{before}},$$

where n is the number of VM invocations, m the number of measurements iterations, j the jth (simulated) benchmark iteration, and i the ith (simulated) VM invocation.

After the termination of all iterations, we collected a set of simulated realizations of the relative performance change $P = \{\rho_i \mid 1 \leq i \leq 1000\}$ and estimated the 0.025 and 0.975 quantiles on it, for a 95% confidence interval. Given a (p, c, b, R) data point, refactoring operations R lead to a *regression* of b, if the lower limit of the confidence interval for relative performance change of mean execution times is greater than 0 (i.e., b becomes slower after the commit c). Similarly, there is an *improvement* in b if the upper limit of the confidence interval is less than 0 (i.e., b is faster before the commit). Otherwise, we consider performance as *unchanged*.

2.2.2 RQ_1: To What Extent Do Developers Refactor Performance-Relevant Code Components? Performance benchmarks usually cover only parts of the system that are relevant to performance [22]. In our study, we consider a Java method as performance relevant if it is covered by at least one benchmark. In other words, given a snapshot of the system c and a method m, we consider m as performance relevant if there exists at least one benchmark b such that $m \in M_b^c$; i.e., m is executed by b. On the other hand, a method is considered non-relevant in terms of performance if it is not covered by any benchmark. To answer RQ_1, we compared the density of refactoring operations in performance-relevant code to the one in other parts of the system. We considered commits where at least one refactoring was detected and for which we were able to collect methods executed by benchmark suites. Table 1 reports, for each project, the number of commits and refactoring operations considered in this RQ. For each commit c we computed:

- PM_c: the number of performance-relevant methods (i.e., methods executed by at least one benchmark) subject to at least one refactoring
- NPM_c: the number of performance-relevant methods not subject to any refactoring operation
- OM_c: the number of methods in the project not executed by any benchmark and subject to at least one refactoring
- NOM_c: the number of methods in the project not executed by any benchmark and not subject to any refactoring

Then, we computed, for every subject system, *refactoring density* in performance-relevant code as the ratio of the number of performance-relevant methods subject to refactoring operations over the total number performance-relevant methods in the entire system history:

$$RDP_C = \frac{\sum_{c \in C} PM_c}{\sum_{c \in C} PM_c + NPM_c},$$

where C is the set of commits under analysis for the subject system. Similarly, we measured refactoring density in code not considered as performance relevant:

$$RDNP_C = \frac{\sum_{c \in C} OM_c}{\sum_{c \in C} OM_c + NOM_c}.$$

It is worth noting that we may count several times the same method if it appears in different snapshots of the system. For example, given a system with two commits, c^1 and c^2, let us consider a performance-relevant method m subject to at least a refactoring operation in both c^1 and c^2.

Table 1. RQ_1

Project	Analyzed Commits (Total)	Refactorings (Total)	Methods (Average)	Performance-relevant Methods (Average)
alibaba/fastjson	34	85	1,249	21
apache/arrow	38	434	2,470	192
apache/camel	327	6,059	53,440	316
apache/commons-bcel	25	98	2,551	204
apache/logging-log4j2	405	1,700	2,161	300
arnaudroger/SimpleFlatMapper	80	1,291	3,148	170
cantaloupe-project/cantaloupe	204	1,297	1,767	121
debezium/debezium	82	417	2,843	107
easymock/objenesis	10	108	88	11
eclipse-ee4j/jersey	28	435	9,338	351
eclipse-vertx/vert.x	139	1,377	4,071	96
eclipse/jetty.project	392	2,218	12,400	86
eclipse/rdf4j	35	240	11,004	737
elastic/apm-agent-java	85	328	1,162	63
HdrHistogram/HdrHistogram	10	55	593	52
iotaledger/iri	29	329	1,220	50
JCTools/JCTools	75	740	870	95
jdbi/jdbi	31	107	1,282	170
jooby-project/jooby	59	425	1,546	7
kiegroup/drools	118	885	26,909	220
netty/netty	88	482	13,223	1,102
OpenFeign/feign	12	50	457	107
OpenHFT/Chronicle-Core	1	1	154	3
openzipkin/zipkin	15	165	1,698	249
panda-lang/panda	145	1,538	1,758	56
prestodb/presto	810	9,558	25,527	464
prometheus/client_java	6	13	264	35
protostuff/protostuff	11	93	1,994	35
pwm-project/pwm	24	560	4,551	6
ReactiveX/RxJava	16	225	3,915	943
zalando/logbook	9	103	513	99

Number of commits and refactoring operations used to evaluate density of refactoring operations. The table also reports (for these commits) the average number of methods in the system and the average number of methods covered by at least one performance benchmark.

Such a method is counted both in PM_{c^1} and in PM_{c^2}; i.e., it is counted twice in the formula of RDP_C. We do this because the same method can have different properties in different commits: it could be counted as PM_{c^1} in a snapshot and as $NPM_{c^2}/OM_{c^2}/NOM_{c^2}$ in another one.

Finally, we report, for systems with more than 50 commits, refactoring density in performance-relevant code as well as refactoring density in other parts of the system via bar charts (Figure 2). We also perform Fisher's exact test [39] to test whether the proportions of $\sum_{c \in C} PM_c / \sum_{c \in C} NPM_c$ and $\sum_{c \in C} OM_c / \sum_{c \in C} NOM_c$ differ significantly.

In addition, we used the Odds Ratio (OR) [39] of the two proportions as effect size measure. An OR of 1 indicates that refactoring performance-relevant code is equally likely as refactoring other parts of the system. An OR greater than 1 indicates that refactoring operations are more likely performed in code non-relevant from a performance perspective. An OR lower than 1 indicates that refactorings are more likely performed in performance-relevant code.

2.2.3 RQ_2: What Is the Impact of Refactoring on Performance? Concerning RQ_2, we need to assess the impact of refactoring-related commits on software performance. In this RQ, we con-

Table 2. RQ_2 and RQ_3

Project	Stars	Commits	Benchmarks
alibaba/fastjson	22,752	3,793	4
apache/arrow	6,684	8,065	47
apache/camel	3,524	49,254	23
apache/logging-log4j2	1,075	11,238	585
cantaloupe-project/cantaloupe	172	4,376	11
eclipse/rdf4j*	229	4,279	132
eclipse-vertx/vert.x	11,552	4,825	41
hazelcast/hazelcast**	4,079	30,670	144
HdrHistogram/HdrHistogram	1,786	740	75
iotaledger/iri	1,183	2,701	3
JCTools/JCTools	2,518	971	172
jgrapht/jgrapht**	1,802	3,185	91
kiegroup/drools	3,356	12,894	1
netty/netty	25,443	10,100	1,686
OpenFeign/feign*	6,471	857	13
prestodb/presto	11,454	18,431	1,534
protostuff/protostuff	1,550	1,580	31
raphw/byte-buddy**	3,904	5,383	39
ReactiveX/RxJava	43,867	5,810	1,302
zalando/logbook	733	1,626	20

Overview of projects considered in the evaluation of the performance impact of refactoring operations (i.e., for which at least one data point was discovered). Projects marked with (*) are only considered in RQ_2, those marked with (**) are only considered in RQ_3.

sider the dataset gathered from the first round of data collection, which involves 1,534 data points, 69 commits, 150 refactoring operations (among 16 refactoring types), and 17 projects (see Table 2 for an overview of the study subjects' projects).

Each refactoring-related commit has one of the following effects on the system performance:

- *Regression*: the commit leads to performance regression of some benchmarks without improving performance of any other benchmark.
- *Improvement*: the commit leads to performance improvement of some benchmarks without worsening performance of any other benchmark.
- *Mixed*: the commit leads to performance regression of some benchmarks and improves the performance of some other benchmark.
- *Unchanged*: the commit keeps the benchmark execution time unmodified.

We report the percentages of refactoring-related commits falling in the four above categories via bar charts in Figure 3. We also report the percentages of data points showing regressions, improvements, or unchanged performance, to evaluate how code affected by refactoring operations is impacted in terms of software performance. In order to provide a comprehensive view on the performance impact of refactoring operations, we also report the magnitude of the performance change for benchmarks showing *regression* and *improvement*. The magnitude of performance change, for a given data point, is measured using the estimated mean relative performance change (i.e., the center of confidence interval; see Section 2.2.1). We depict the distribution of these means via box plots for both data points showing regressions and data points showing improvements in Figure 5.

2.2.4 RQ_3: What Types of Refactoring Operations Are More Likely to Impact Performance? To answer RQ_3, we need to isolate the effect of different refactoring types on software performance.

We selected from our dataset the data points (p, c, b, R) having all refactoring operations $r \in R$ of the same type. Types of refactoring with less than 50 associated data points are excluded from this analysis. We analyzed 1,156 data points from 18 systems (see Table 2) involving seven refactoring types: Extract Method (166 data points), Extract Superclass (90), Inline Variable (65), Extract Class (398), Move Method (184), Inline Method (66), and Extract Interface (187).

To analyze the impact on software performance of each refactoring type, we report via bar charts the percentage of data points showing regression, improvement, and unchanged performance (see Figure 6). We also report the magnitude of regressions and improvements for each type of refactoring via box plots (see Figures 7 and 8). In the latter analysis we discarded types having negligible impact in terms of both regression and improvement, i.e., those that have less than 5% associated data points showing regressions or improvements. Finally, we discuss interesting examples related to different types of refactoring operations.

2.2.5 Qualitative Analysis. To better understand how and why refactoring operations impact the performance, five of the authors manually inspected commits, issues, and pull requests related to cases in which refactoring had a negative impact on execution time. We report interesting cases and discuss them along with RQ_2 and RQ_3 results.

2.3 Replication Package

We provide in our replication package [44] the complete data needed to replicate our findings. In particular, we share the SHA code of the subject commits from each of the analyzed systems, together with the refactoring operations detected in them and the results of the benchmarks' execution. We also provide the *Python* scripts used to generate the figures and tables reported in Section 3.

3 RESULTS DISCUSSION

3.1 RQ_1: To What Extent Do Developers Refactor Performance-Relevant Code Components?

Figure 2 reports the density of refactoring operations in performance-relevant and non-performance-relevant methods by software system. The darker bars represent refactoring density in performance-relevant methods, i.e., the chance that a performance-relevant method is subject to refactoring. Similarly, the lighter bars represent refactoring density in other methods of the system (i.e., non-performance relevant).

As previously explained, we group all systems for which we collected less than 50 commits relevant for RQ_1 in "others" (bottom of Figure 2). The first observation that can be made by looking at Figure 2 is that no matter whether the method is performance relevant or not, the chance that it is subject to refactoring operations is quite low (i.e., less than 1.5%—all bar charts are below the 0.015 on the x axis).

When comparing the refactoring density in performance-relevant methods with that of non-performance-relevant methods, interesting trends can be observed. In only two projects (i.e., `camel` and `drools`), developers performed more refactoring operations on performance-relevant methods. However, according to Fisher's exact test, only in one project (i.e., `camel`) is such a difference statistically significant (p-value < 0.05) with an OR of 0.37.

For all other projects, the refactoring density is higher in non-performance-relevant methods. Among those, only two projects (i.e., `jooby` and `vert.x`) have a p-value larger than 0.05 (i.e., the difference is not statistically significant). For all other projects, the refactoring density difference is statistically significant, with ORs varying from 1.47 to 11.19. This result indicates that in most projects, the density of refactoring operations is higher in non-performance-relevant methods.

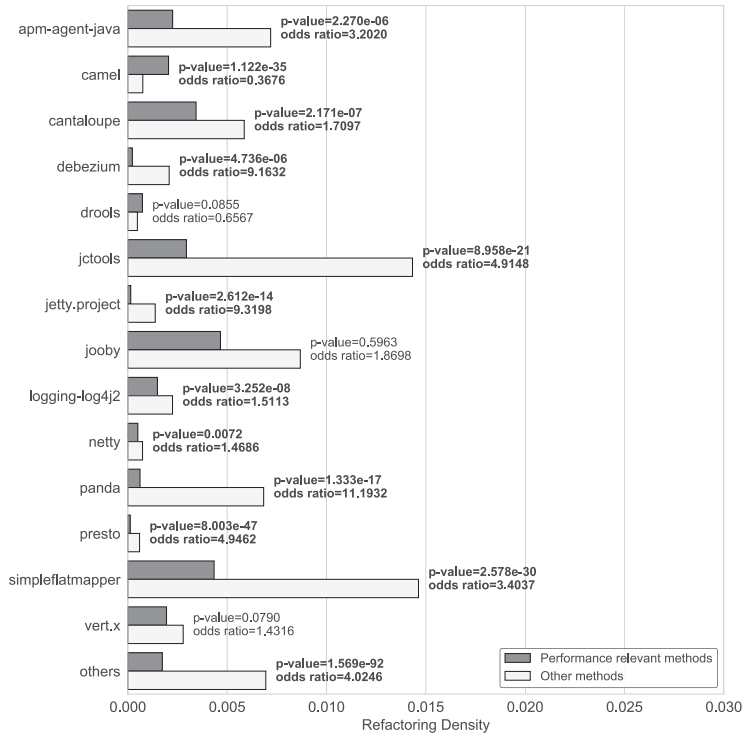

Fig. 2. RQ_1. Density of refactoring operations in performance-relevant methods and in other methods. Fisher's exact test results accompanied with Odds Ratio are also reported.

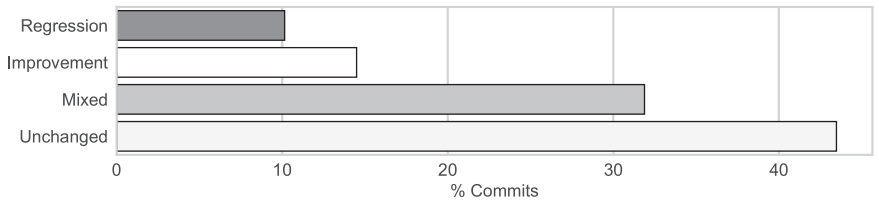

Fig. 3. RQ_2. Percentages of refactoring-related commits leading to regression, improvement, mixed effect, or unchanged execution time.

We conjecture that the potential performance impact might be one of the factors that discourages developers from refactoring performance-relevant methods. Validating such a conjecture would require a dedicated empirical study surveying developers. While this is out of the scope of this work, we proceed in the following RQs with investigating the impact on the execution time of the refactoring operations that focused on performance-relevant methods.

3.2 RQ_2: What Is the Impact of Refactoring on Performance?

Figure 3 shows the impact of refactoring-related commits on execution time.

It is worth noting that multiple benchmarks can be involved in each commit. As can be seen from the chart, more than 40% of refactoring-related commits do not result in any performance change. In the rest of the cases, a large percentage of the commits (>30%) have a mixed effect on performance, *in that on the same commit some benchmarks induce performance*

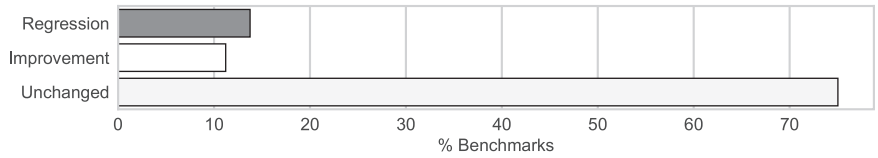

Fig. 4. RQ_2. Percentages of benchmarks affected by refactoring-related commits (i.e., data points) showing regressions, improvements, or unchanged execution time.

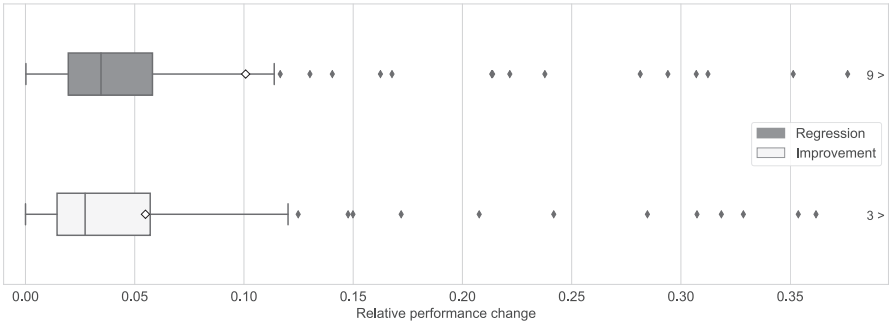

Fig. 5. RQ_2. Relative performance change of benchmarks due to refactoring operations. The darker box plot reports results for benchmarks showing regression, while the lighter box plot reports results for benchmarks showing improvement.

regression, whereas some other ones induce improvement. Around 15% of the commits only lead to performance improvement, and slightly more than 10% cause only performance regression. This is not surprising as code affected by refactoring operations can be exercised in different ways by benchmarks. For example, each benchmark may involve the execution of a different set of performance-relevant methods or the execution of the same methods with different inputs, which can lead to diverse performance behaviors. From these results, we can observe that a large percentage of the commits (>55%) (i.e., the ones that are not classified as unchanged) cause a statistically significant performance change (either positive or negative) on methods of the system that are considered relevant for performance. Specifically, when considering the negative impact on performance due to refactorings, more than 40% of the commits cause performance regression in at least one benchmark. This is particularly relevant considering that performance issues are usually discovered through specific tests and inputs [18, 27, 43].

By inspecting how the performance is impacted in each benchmark affected by refactoring-related commits (Figure 4), we can find that in more than 75% of cases, the performance is not changed.

Neither performance regression nor performance improvement is common, and they both take place in around 10% of the benchmarks. This suggests that, similarly to common performance issues [18], performance changes introduced by refactoring operations require specific benchmarks to be exposed [27, 43]. That is, even when the commit causes a performance change in some of the benchmarks, there are often other benchmarks for which performance remains unchanged.

We further looked into the extent of performance regression or improvement (Figure 5), and we can find that the medians of relative performance changes are below 5% for both performance improvement and regression.

About 50% of regressions involve performance changes between 2% and 6%, and, similarly, 50% of improvements range between 1% and 6%. Moreover, it is rare that refactoring-related commits can lead to a performance change of more than 15%. This is expected considering that we are

investigating code changes performed in a single code commit. The magnitude of these changes, which may appear negligible, is still relevant as benchmarks measure performance at the method level [22–24]. Indeed, even a relatively small performance change at the method level may potentially lead to a huge performance deviation at the system level.

For example, in the pull request 8614[8] of `netty/netty` we found that a code refactoring was not accepted as it causes "non-negligible" performance regression (i.e., up to 5%) in two benchmarks. This confirms that performance changes due to refactoring operations (see Figure 4) may be relevant for performance.

The pull request 8614, mentioned above, also highlights interesting aspects about the relationships between refactoring activities and software performance. The goal of this pull request is to achieve "less code duplication" and "better encapsulation" by removing duplicated logic from two classes with the help of a common helper class, and it involves several refactoring operations such as Extract Class, Change Parameter Type, and Move Method. After the performance regressions were detected in the two benchmarks, the developer revised the code changes and eliminated the regression, and finally the pull request was merged. Nevertheless, while our benchmark results show no regression on the benchmarks used by developers (which is in line with developer expectations), we did find significant performance regressions (up to 3 times) on other benchmarks not considered by developers. This may suggest that *comprehensively analyzing the performance impact of refactoring operations is not trivial, and even experienced developers might consider only a part of it.*

Another interesting fact is that although some projects attach great importance to the performance, they merge refactoring-related commits without verifying their performance impact. For example, two commits (`49ac2da`[9] and `f537eda`[10]) performing "Extract Superclass" operations were proposed in the same pull request 185[11] for the project `JCTools/JCTools`, in order to "homogenize atomic queues" (i.e., making the atomic queue class `AtomicArrayQueue` as similar to the unsafe queue `ArrayQueue` as possible). While the author expressed concerns about performance ("Performance impact of this is unverified"), the pull request was merged without any discussion. Nevertheless, we found that these commits have non-negligible impact on performance (up to 11%). We conjecture that the long execution time required to run performance benchmark suites (e.g., more than 2 hours for `JCTools/JCTools` [22]) may prevent developers from verifying the performance impact of refactoring operations. The adoption of state-of-the-art techniques [24] to reduce benchmarks' execution time without sacrificing result quality may be beneficial for this problem. Also, similarly to what has been done to predict the impact of a refactoring operation on quality metrics before applying it [6], it might be beneficial to design techniques able to predict the impact of refactoring operations on performance.

Finally, since our analyses have been conducted at commit-level granularity and a commit can involve multiple refactoring operations, we also investigated whether a correlation exists between the number of refactoring operations in a commit and the magnitude of the change in performance. We computed the Spearman rank correlation coefficient, which reported a lack of correlation between the number of refactorings in the commits and performance change ($\rho = 0.118$). This is somehow expected when assessing the impact of refactoring commits grouping together different types of refactoring, as done in RQ_2. Indeed, some of them may bring performance improvements, while others may bring regressions, thus not showing a clear trend.

[8] https://github.com/netty/netty/pull/8614.
[9] http://bit.ly/3oRLfza.
[10] http://bit.ly/34aqgjd.
[11] https://github.com/JCTools/JCTools/pull/185.

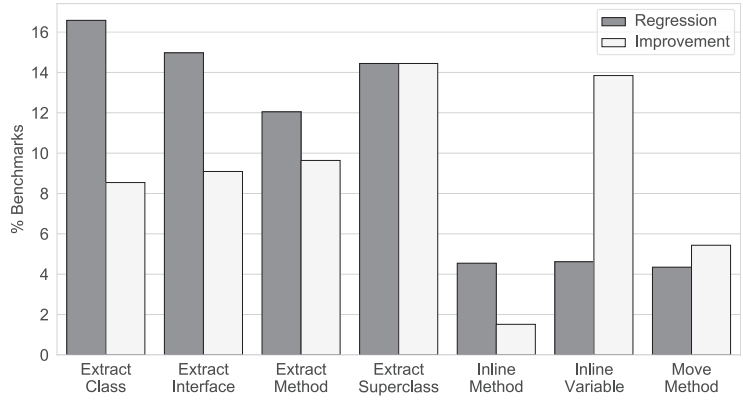

Fig. 6. RQ_3. Performance impact of different types of refactoring on the associated benchmarks (i.e., data points). Percentages of benchmarks showing regression or improvement are reported for each refactoring type.

Summing up, most refactoring-related commits lead to performance change, with these changes usually affecting only a subset of the involved benchmarks. Moreover, we found that a large percent of commits (>55%) lead to regression in at least one benchmark. Performance regressions and improvements due to refactoring-related commits have relatively similar frequencies, and they can bring a performance change up to 12% in most of the cases. Nevertheless, these relatively small performance changes may still be relevant for system-level performance, especially in case of methods involved in "core features." Finally, our results indicate that the analysis of the performance impact of refactoring activities may be non-trivial even for experienced developers, as these changes can have diverse (and often mixed) effects on performance-relevant methods. This problem is further exacerbated by the long execution time required to run benchmark suites, which may prevent developers from verifying the performance impact of their refactoring operations.

3.3 RQ_3: What Types of Refactoring Operations Are More Likely to Impact Performance?

Figure 6 reports the percentage of benchmarks in which the performance is positively or negatively impacted by each type of refactoring operation considered in RQ_3.

The chart reveals that all of the refactoring types can lead to both improved and regressed performance. Overall, Extract Class/Interface/Method/Superclass refactoring operations are more likely to impact performance than Inline Method/Variable and Move Method. When performing Extract Class/Interface/Method and Inline Method, the performance is more likely to degrade, while when performing Inline Variable and Move Method, there is a higher chance of performance improvement. Extract Superclass leads to similar amounts of performance regression and improvement.

The Extract Class refactoring is the more closely related to performance regression, with more than 16% of impacting benchmarks showing such a trend. Moreover, the magnitude of regression introduced by Extract Class is higher when compared to other types of refactorings (50% of regressions lead to a performance change between 2% and 7%; see Figure 7).

Indeed, Extract Class refactoring may induce a higher number of allocated objects, which may regress the performance of the system. For example, the description of the issue `LOG4J2-1295`[12] of `apache/logging-log4j2` states, "*it is not always obvious that some code creates objects, so*

[12] https://issues.apache.org/jira/browse/LOG4J2-1295.

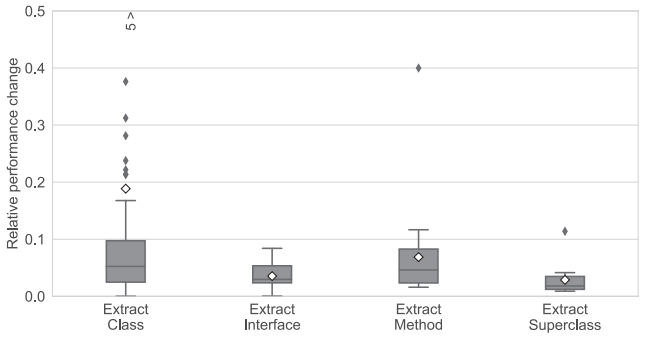

Fig. 7. RQ_3. Relative performance change of benchmarks showing regressions due to different types of refactoring operations.

it is easy for regressions to creep in during maintenance code changes." Indeed, according to `apache/logging-log4j2` developers,[13] "*garbage collection (GC) pauses are a common cause of latency spikes*" and the allocation of more temporary objects "*contributes to pressure on the garbage collector and increases the frequency with which GC pauses occur.*" Therefore, when Extract Class refactoring causes a higher number of allocated objects, it may increase the frequency of GC pauses, thereby leading to performance regression. Another interesting fact is that even the same Extract Class operation can have significantly different performance impact on slightly different versions of software. For example, when we inspected a case of non-negligible performance regression (i.e., performance reduced 8% and 20% for two benchmarks, respectively) caused by Extract Class in the commit `90d82d2`[14] of `apache/logging-log4j2`, we found the same refactoring operation was performed in another branch of the same project[15] (`9ad3603`[16]). However, this refactoring only causes regression of up to 5% for seven different benchmarks. This finding suggests that even the same Extract Class operation can have different impact on performance under different contexts.

Extract Interface, Extract Superclass, and Extract Method also have higher chances to lead to performance regressions compared to other refactorings (respectively, ~15%, ~14%, and ~12% of the involved benchmarks show regression). While the former two cause less intense regressions, Extract Method provides regressions with similar magnitudes to those observed for Extract Class (see Figure 7). Although improvements due to Extract Method are less frequent than regressions, they lead to higher performance changes (50% of them range from 2% to 14%; see Figure 8).

This behavior relies on the relationship between Extract Method and specific runtime optimizations employed by the JVM, as smaller methods have more chances to be inlined during runtime optimization of a Just-In-Time (JIT) compiler. Extract Method refactoring is common practice to achieve automatic inlining at runtime. It is often difficult to identify such optimization opportunities as they require specific conditions. To become a candidate for inlining, a method must satisfy at least one of two conditions: (1) its bytecode size must be within 35 bytes (by default), or (2) it must be called more often than a pre-defined threshold (10,000 invocations by default) and its bytecode size must be within 325 bytes (by default) [32]. Usual benchmark configurations are ineffective to identify such optimization opportunities. In commit `ceb0a62`[17] of

[13] http://bit.ly/3apTONJ.
[14] http://bit.ly/2WnBOv3.
[15] Note that, according to our manual analysis, this is the only case in which the same commit was performed in multiple branches of the system.
[16] http://bit.ly/38cVpnd.
[17] http://bit.ly/3gXtiN2.

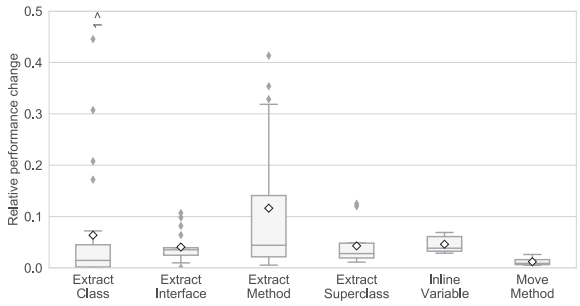

Fig. 8. RQ_3. Relative performance change of benchmarks due to different types of refactoring operations.

`apache/logging-log4j2`, which "*refactors a large method into smaller methods to enable inlining*," improvement is expected in execution time as these smaller methods have more chances to be inlined. Our benchmark results only displayed performance regressions. This kind of optimization may not manifest when evaluating performance through default benchmarking configurations, as they are triggered only in specific scenarios (the authors mentioned in the commit message that "*the new code is all inlined after ~7000 invocations*"). We envision that future research may leverage static characteristics of the code (e.g., size of methods) combined with its dynamic behavior (e.g., the number method invocations during benchmarking) to design recommenders that automatically suggest potential optimization opportunities through Extract Method refactoring.

For the other types of refactoring, Inline Variable has a relatively high chance to lead to performance improvement (14% of the benchmarks with a performance change ranging from 3% to 6% in 50% of the cases), while Move Method and Inline Method have lower chances to bring performance change. Moreover, the performance change caused by the Move Method has never reached 5%.

As already done for RQ_2, also in this case we investigate a possible correlation between the number of refactoring operations in a commit and the magnitude of the change in performance. We did this analysis by refactoring type. Extract Method and Extract Class were the two showing the strongest Spearman rank correlation coefficient, which, however, still showed a poor correlation between number of refactorings in a commit and performance change ($\rho = 0.213$ for Extract Method and $\rho = 0.108$ for Extract Class).

In summary, the impact of refactoring on performance varies from type to type. No refactoring type guarantees the absence of performance regression. Extract Class and Extract Method have a higher chance of causing larger performance regression than other types of refactoring. When Extract Method causes performance improvement, it leads to larger performance changes.

4 THREATS TO VALIDITY

Construct validity. The main threats to the construct validity of our study are related to the process we adopted to measure performance variations caused by refactoring.

To mitigate the risk of unstable performance benchmark results, we perform, within each VM invocation, multiple warmup and measurement iterations according to the JMH configuration defined by software developers, and we fix the number of VM invocations to 10 as done in previous studies [13, 23]. We did not use developer custom configurations for VM invocations, since a previous study [24] showed that developers often rely on a single VM invocation, which is considered a bad practice as inter-JVM-variability is common [13, 19, 22]. Using configurations with a higher number of iterations or VM invocations may lead to more stable results. Prior work [13] suggests to dynamically stop measurement iterations when certain quality criteria are met

(e.g., coefficient of variation < 0.02). Nevertheless, we found this approach impractical for our study due to extremely long execution times. Software compilation may also induce performance variability [31] due to the non-deterministic nature of Java compilation strategies [14, 19]. Such variability can be mitigated through compiler replay [14, 19] to avoid bias introduced by compilation. However, these approaches can dramatically increase the time needed for benchmarking as they add another level of repetition, which makes them impractical for our study. To reliably detect performance change, while dealing with performance variability, we followed best practices [5, 19, 23, 24]. We estimate the confidence interval for relative performance change with bootstrap [10, 20] by employing hierarchical random resampling with replacement [34], and we detect performance change if there is statistically significant difference—i.e., the confidence interval does not contain 0.

Imprecisions in the detected refactorings could also have affected our results. However, we used a highly precise state-of-the-art tool (RefactoringMiner [46]), reported to have a 98% precision and 87% recall. Also, while it is possible that we missed relevant data points for our study due to false negatives (i.e., refactoring-related commits missed by RefactoringMiner), we are confident about the absence of false positives in our dataset, since we manually inspected all the commits subject of our study to exclude those introducing, besides the detected refactorings, other code changes.

Conclusion validity. Wherever possible we used appropriate statistical procedures with p-value and effect size measures to test the significance of the differences and their magnitude.

Internal validity. Those are mainly related to a missing causation link between refactorings and changes in performance as assessed by the benchmarks, and to possible confounding factors that may influence such a relationship. We controlled for tangled commits, ensuring that the commits considered in our study only focused on refactoring-related changes. However, (1) in our observational study we do not claim causation, and (2) at least, we complemented the quantitative analysis with a qualitative one, which helped in better understanding the influence of refactoring on performance.

External validity. While the number of systems and the subject commits are limited as compared to those of MSR studies, it is worth nothing that the data collection procedure for our study required 15 months of work. Moreover, the number of systems we consider is larger than recent studies investigating software performance research questions (see, e.g., [12, 22, 24, 33]), while the numbers of commits and performance tests involved are similar. Still, the generalizability of our findings is limited to the analyzed refactoring types and systems.

5 RELATED WORK

Given the goal of our study, we discuss the literature related to studies investigating (1) software performance in the context of code evolution and (2) the impact of refactoring operations on quality attributes.

5.1 Empirical Studies Relating Performance to Software Evolution

Performance analysis of running software systems has been tackled by different perspectives in the last few years (e.g., through models at runtime [2, 15]). However, given the goal of our article, we focus here on the domain of empirical analyses, possibly supported by benchmark techniques.

Han et al. [16] have introduced StackMine, a tool exploiting stack traces to allow performance debugging of a considerable amount of data on Windows-based systems. Our approach works at a higher level of abstraction because, on the basis of traces generated through JMH microbenchmarks, we aim at identifying the (beneficial or degrading) effects of refactoring actions on software performance.

Sandoval et al. [36–38] have analyzed performance regression of different versions of applications in an object-oriented language and development environment named Pharo.[18] They have analyzed 19 different projects and a total of 1,125 different versions. The main differences between our approach and the one in [36–38] are first, the context (i.e., Java systems *vs* Pharo), and second, we have exploited JMH as a micro-benchmarking library instead of building an in-house benchmark suite. Furthermore, we have analyzed 20 different open-source projects by generating 1,598 data points.

Daly et al. [9] have presented mechanisms for detecting performance regression in an industrial project, i.e., MongoDB. They automatically detect change-points' variability to identify the commit causing a specific performance degradation event. Then, those labeled points have been manually checked to discard false positives. Our process starts from commits labeled with specific refactoring actions and, then, look at their effect on performance. Also, our study spans different projects.

Laaber et al. [24] have focused their study on reducing the required execution time of micro-benchmarking tests through a dynamic reconfiguration of JMH. They have defined three ways to detect when a test reaches the performance peak (i.e., the steady state) and then they apply their reconfigurations. In our study, it would be interesting to use the approach proposed by Laaber et al. with the aim of reducing the duration of our tests. However, we have decided to exploit the default JMH configurations (i.e., the ones associated to the different commits) to be as compliant as possible with developers intents.

Chen et al. [7] have studied the influence of code changes on performance degradation in the context of the Python programming language. They have exploited unit tests, along with a profiler, to extract performance data. Our study design differs from the one by Chen et al., since we target Java programming language and exploit micro-benchmarks, instead of unit tests, to extract performance data. Also, we focus on a specific type of code changes (i.e., refactoring actions).

Reichelt et al. [33] have compared unit tests to discover performance regression between versions of nine long-lived Java open-source projects. They use such a corpus to infer performance variations through code changes. We rely on JMH instead of unit tests, because the former avoids JVM optimizations that may produce unreliable performance data.

Ding et al. [12] have analyzed whether unit tests can be aimed at assessing performance. In particular, they have targeted two systems (i.e., Cassandra[19] and Hadoop[20]), and they have extracted functional tests that can be performance related by digging developers' message backlogs. We have instead dug the GitHub corpus in order to extract projects equipped with JMH tests, and we have investigated the correlation between refactoring actions and performance degradation in 20 systems.

Table 3 lists the sizes of corpora of related works that empirically assess software performance. To avoid second-guessing, we only report such data for studies explicitly providing information about the number of subject projects and benchmarks.

To the best of our knowledge, the magnitude of our study is on par with, if not larger than, previous works empirically analyzing changes in performance caused by source code changes.

5.2 On the Impact of Refactoring on Code Quality Attributes

In recent years, many researchers have focused on how refactorings might impact the quality of software projects.

[18] Pharo project, http://pharo.org.
[19] Cassandra, https://cassandra.apache.org/.
[20] Hadoop, https://hadoop.apache.org/.

Table 3. Comparison among Corpora Sizes

Reference	Projects	Benchmarks
Sandoval et al. [36–38]	19	1,125
Laaber et al. [24]	10	2,164
Chen et al. [7]	8	1,268
Reichelt et al. [33]	9	105
Our study	20	1,598

Moser et al. [30] conducted a case study on a project developed in an agile and close-to-industrial environment. The authors examined the code quality change after refactorings, with complexity and coupling metrics. They found that refactorings lead to simpler and less coupled code.

Szőke et al. [41] analyzed five software systems and measured the quality change over refactorings with a probabilistic quality model. With the 200 identified refactoring commits, the authors found that while single refactoring does not necessarily increase the software quality, its increase in local components and globally can be more evident when refactorings are applied in blocks.

Tavares et al. [42] applied 80 refactorings automatically generated by JDeodorant on seven open-source Java systems and investigated how refactoring impacts code smells. Their results indicate that while some code smells can be eliminated by refactoring as expected, there are also cases in which refactorings introduce new bad smells.

Abid et al. [1] examined the impact of refactorings on both security and other quality attributes (i.e., reusability, flexibility, understandability, functionality, extendibility, and effectiveness). By analyzing 30 open-source software projects, they found that while refactorings help to improve other quality attributes, the software tends to become less secure. This negative correlation needs to be taken into account before refactoring software systems.

Lin et al. [26] inspected 1,448 refactoring operations from 619 Java projects to understand whether refactorings lead to more natural code, namely whether the source code becomes more repetitive and predictable. Their results indicate that this assumption does not always hold, and the impact on the code naturalness varies among different types of refactorings.

Sahin et al. [35] conducted an empirical study, involving 197 applications of six commonly used refactorings, to investigate how refactorings affect the energy usage. Their results show that all the considered refactorings in the study can potentially impact the energy consumption, with a magnitude ranging from −4.6% to 7.5%. Verdecchia et al. [47] also looked into the same topic. They applied automatic refactoring on five different types of code smells in three open-source Java projects and collected energy consumption in a controlled environment. As a result, they found that in one project, refactoring significantly impacted the energy consumption.

To the best of our knowledge, not many studies have investigated the impact of refactoring operations on the performance of software systems. The most relevant work to ours is the study conducted by Demeyer [11], which inspected how a specific type of refactoring (i.e., replacing conditionals by virtual function calls) impacts the performance of C++ programs. Their results show that this type of refactoring often leads to faster performance compared to their non-refactored counterparts. While this study is highly relevant to software performance, it only focuses on a specific type of refactoring operation.

6 CONCLUSION

We presented an empirical study aimed at investigating the impact of refactoring operations on execution time. To the best of our knowledge, this is the first work analyzing a wide set of refactoring types from the "performance perspective." As for any performance-related study, the collection of

the data needed to answer our research questions posed major challenges and required hundreds of machine days.

The achieved results show that the impact of refactoring on execution time varies depending on the refactoring type, with none of them being 100% "safe" in ensuring that there is no performance regression. Some refactoring types, such as Extract Class and Extract Method, can result in substantial performance regression and, as such, should be carefully considered when refactoring performance-critical parts of a system. It is important to highlight that, due to the expensive data collection process behind our study, such results are based on a limited number of data points (e.g., a total of 82 commits from 20 projects). Additional investigations are needed to strengthen the generalizability of our findings. Still, our work discloses the potential side effects of refactoring on execution time and has implications for both practitioners and researchers. For the former, it is important to be aware of the possible performance regressions caused by refactoring operations. The recommendation here is not to avoid refactoring performance-critical code but to properly handle it. First, developers should ensure that the code target of the refactoring is properly covered by performance benchmarks. In this way "performance regression testing" can be performed after refactoring to assess if and how much the implemented changes degraded performance. Second, assuming that a cost is observed in terms of performance, a non-trivial cost-benefit analysis must be run to decide whether to revert the refactoring. For example, if the refactored code is well known to cause maintainability issues (e.g., it is difficult to comprehend, leading to frequent bug introduction), developers may accept some performance regression to improve code maintainability. On the researchers' side, we envision research aimed at designing (1) approaches to predict the impact on performance of planned refactoring operations before they are actually implemented in the system and (2) sensible refactoring recommender systems able to consider tradeoffs between multiple non-functional requirements when making recommendations. Our future agenda is driven by these two research directions.

REFERENCES

[1] C. Abid, M. Kessentini, V. Alizadeh, M. Dhouadi, and R. Kazman. 2020. How does refactoring impact security when improving quality? A security-aware refactoring approach. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.3005995

[2] Davide Arcelli, Vittorio Cortellessa, Daniele Di Pompeo, Romina Eramo, and Michele Tucci. 2019. Exploiting architecture/runtime model-driven traceability for performance improvement. In *IEEE International Conference on Software Architecture (ICSA'19)*. IEEE, 81–90. https://doi.org/10.1109/ICSA.2019.00017

[3] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*. Springer, 387–419. https://doi.org/10.1007/978-3-642-45135-5_15

[4] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2014. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering (TSE)* 40, 7 (2014), 671–694.

[5] Lubomír Bulej, Vojtěch Horký, Petr Tuma, François Farquet, and Aleksandar Prokopec. 2020. Duet benchmarking: Improving measurement accuracy in the cloud. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE'20)*. Association for Computing Machinery, New York, NY, 100–107. https://doi.org/10.1145/3358960.3379132

[6] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. 2014. On the impact of refactoring operations on code quality metrics. In *30th IEEE International Conference on Software Maintenance and Evolution*. 456–460.

[7] Jie Chen, Dongjin Yu, Haiyang Hu, Zhongjin Li, and Hua Hu. 2019. Analyzing performance-aware code changes in software development process. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC'19)*, Yann-Gaël Guéhéneuc, Foutse Khomh, and Federica Sarro (Eds.). IEEE/ACM, 300–310. https://doi.org/10.1109/ICPC.2019.00049

[8] Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. 2011. Lightweight transformation and fact extraction with the srcML toolkit. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 173–184.

[9] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. 2020. The use of change point detection to identify software performance regressions in a continuous integration system. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE'20)*. Association for Computing Machinery, New York, NY, 67–75. https://doi.org/10.1145/3358960.3375791

[10] A. C. Davison and D. V. Hinkley. 1997. *Bootstrap Methods and their Application*. Cambridge University Press. https://doi.org/10.1017/CBO9780511802843

[11] S. Demeyer. 2005. Refactor conditionals into polymorphism: What's the performance cost of introducing virtual calls? In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 627–630.

[12] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet? In *42nd International Conference on Software Engineering (ICSE'20)*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1435–1446. https://doi.org/10.1145/3377811.3380351

[13] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*. Association for Computing Machinery, New York, NY, 57–76. https://doi.org/10.1145/1297027.1297033

[14] Andy Georges, Lieven Eeckhout, and Dries Buytaert. 2008. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'08)*. Association for Computing Machinery, New York, NY, 367–384. https://doi.org/10.1145/1449764.1449794

[15] Holger Giese, Leen Lambers, and Christian Zöllner. 2020. From classic to agile: Experiences from more than a decade of project-based modeling education. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELs'20), Virtual Event, Companion Proceedings*, Esther Guerra and Ludovico Iovino (Eds.). ACM, 22:1–22:10. https://doi.org/10.1145/3417990.3418743

[16] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance debugging in the large via mining millions of stack traces. In *2012 34th International Conference on Software Engineering (ICSE'12)*. 145–155. https://doi.org/10.1109/ICSE.2012.6227198 ZSCC: 0000165 ISSN: 1558-1225.

[17] K. Herzig and A. Zeller. 2013. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR'13)*. 121–130.

[18] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *SIGPLAN Notices* 47, 6 (June 2012), 77–88. https://doi.org/10.1145/2345156.2254075

[19] Tomas Kalibera and Richard Jones. 2013. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM'13)*. Association for Computing Machinery, New York, NY, 63–74. https://doi.org/10.1145/2491894.2464160

[20] Tomas Kalibera and Richard Jones. 2020. Quantifying Performance Changes with Effect Size Confidence Intervals. (2020). arXiv:stat.ME/2007.10899.

[21] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.

[22] Christoph Laaber and Philipp Leitner. 2018. An evaluation of open-source software microbenchmark suites for continuous performance assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*. Association for Computing Machinery, New York, NY, 119–130. https://doi.org/10.1145/3196398.3196407

[23] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software microbenchmarking in the cloud. How bad is it really? *Empirical Software Engineering* 24, 4 (Aug. 2019), 2469–2508. https://doi.org/10.1007/s10664-019-09681-1

[24] Christoph Laaber, Stefan Würsten, Harald C. Gall, and Philipp Leitner. 2020. Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality. In *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20), Virtual Event*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 989–1001. https://doi.org/10.1145/3368089.3409683

[25] Philipp Leitner and Cor-Paul Bezemer. 2017. An exploratory study of the state of practice of performance testing in Java-based open source projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE'17)*. Association for Computing Machinery, New York, NY, 373–384. https://doi.org/10.1145/3030207.3030213

[26] Bin Lin, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. On the impact of refactoring operations on code naturalness. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*. IEEE, 594–598.

[27] Qi Luo, Aswathy Nair, Mark Grechanik, and Denys Poshyvanyk. 2017. FOREPOST: Finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering* 22, 1 (2017), 6–56. https://doi.org/10.1007/s10664-015-9413-5

[28] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. 2015. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 3 (May 2015), 17:1–17:45.

[29] Rodrigo Morales, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. 2018. EARMO: An energy-aware refactoring approach for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 59.

[30] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. 2007. A case study on the impact of refactoring on quality and productivity in an agile team. In *Proceedings of the 2nd IFIP Central and East European Conference on Software Engineering Techniques (CEE-SET'07)*. Springer, 252–266.

[31] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing wrong data without doing anything obviously wrong! *SIGPLAN Notices* 44, 3 (March 2009), 265–276. https://doi.org/10.1145/1508284.1508275

[32] Scott Oaks. 2014. *Java Performance: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[33] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. 2019. PeASS: A tool for identifying performance changes at code level. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, 1146–1149. https://doi.org/10.1109/ASE.2019.00123

[34] Shiquan Ren, Hong Lai, Wenjing Tong, Mostafa Aminzadeh, Xuezhang Hou, and Shenghan Lai. 2010. Nonparametric bootstrapping for hierarchical data. *Journal of Applied Statistics* 37, 9 (2010), 1487–1498. https://doi.org/10.1080/02664760903046102 arXiv:https://doi.org/10.1080/02664760903046102

[35] Cagri Sahin, Lori Pollock, and James Clause. 2014. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*. Article 36, 36:1–36:10 pages.

[36] Juan Pablo Sandoval Alcocer, Fabian Beck, and Alexandre Bergel. 2019. Performance evolution matrix: Visualizing performance variations along software versions. In *2019 Working Conference on Software Visualization (VISSOFT'19)*. 1–11. https://doi.org/10.1109/VISSOFT.2019.00009 ZSCC: 0000001.

[37] Juan Pablo Sandoval Alcocer and Alexandre Bergel. 2015. Tracking down performance variation against source code evolution. *ACM SIGPLAN Notices* 51, 2 (2015), 129–139. https://doi.org/10.1145/2936313.2816718 Number: 2 ZSCC: 0000013.

[38] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE'16)*. Association for Computing Machinery, 37–48. https://doi.org/10.1145/2851553.2851571 ZSCC: 0000024.

[39] David J. Sheskin. 2007. *Handbook of Parametric and Nonparametric Statistical Procedures* (4th ed.). Chapman & Hall/CRC.

[40] Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. 2017. Unit testing performance in java projects: Are we there yet? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE'17)*. Association for Computing Machinery, New York, NY, 401–412. https://doi.org/10.1145/3030207.3030226

[41] Gábor Szóke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2014. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*. IEEE, 95–104.

[42] Cleiton Tavares, Mariza A. S. Bigonha, and Eduardo Figueiredo. 2020. Quantifying the effects of refactorings on bad smells. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering (SBES'20)*.

[43] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2018. Synthesizing programs that expose performance bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO'18)*. Association for Computing Machinery, New York, NY, 314–326. https://doi.org/10.1145/3168830

[44] Luca Traini, Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. 2020. How Software Refactoring Impacts Execution Time - Replication Package. https://github.com/SEALABQualityGroup/replicationpackage_refperf.

[45] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering (TSE)* 35, 3 (2009), 347–367.

[46] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2020), 21. https://doi.org/10.1109/TSE.2020.3007722

[47] Roberto Verdecchia, René Aparicio Saez, Giuseppe Procaccianti, and Patricia Lago. 2018. Empirical evaluation of the energy impact of refactoring code smells. In *Proceedings of the 5th International Conference on Information and Communication Technology for Sustainability (ICT4S'18) (EPiC Series in Computing)*, Vol. 52. 365–383.

Received December 2020; revised June 2021; accepted August 2021