A Large-Scale Empirical Study of Real-Life Performance Issues in Open Source Projects

Yutong Zhao, Lu Xiao, Andre B. Bondi, Bihuan Chen, *Member, IEEE*, and Yang Liu, *Senior Member, IEEE*

Abstract—Software performance is a critical quality attribute that determines the success of a software system. However, practitioners lack comprehensive and holistic understanding of how real-life performance issues are caused and resolved in practice from the technical, engineering, and economic perspectives. This paper presents a large-scale empirical study of 570 real-life performance issues from 13 open source projects from various problem domains, and implemented in three popular programming languages, Java (192 issues), C/C++ (162 issues), and Python (216 issues). From the technical perspective, we summarize *eight* general types of performance issues with corresponding root causes and resolutions that apply for all three languages. We also identify available tools for detecting and resolving different types of issues from the literature. In addition, we found that 27% of the 570 issues are resolved by design-level optimization—coordinated revision of a group of related source files and their design structure. We reveal four typical design-level optimization patterns, including *classic design patterns*, *change propagation*, *optimization clone*, and *parallel optimization* that practitioners should be aware of in resolving performance issues. From the engineering perspective, this study analyzes how test code changes in performance optimization. We found that only 15% of the 570 performance issues involve revision of test code. In most cases, the revised test cases focus on the functional logic of the performance optimization, rather than directly evaluate the performance improvement. This finding points to the potential lack of engineering standard for formally verifying performance optimization in regression testing. Finally, from the economic perspective, we analyze the *"Return On Investment"* of performance optimization. We found that design-level optimization usually requires more investment, but not always yields to higher performance improvement. However, developers tend to use design-level optimization when they concern about other quality attributes, such as maintainability and readability.

Index Terms—Design patterns, software design structure, software performance, testing code

1 Introduction

Software performance is a critical quality attribute measured by the timeliness, responsiveness, and resource consumption of a system at run-time [1], [2], [3], [4], [5]. Performance issues can lead to severe consequences, including budget overrun, project delay, and market loss [1], [4]. Despite the numerous prior studies that investigate the causes and resolutions of real-life performance issues [4], [6], [7], [8], [9], practitioners still lack a comprehensive and holistic understanding of how performance issues are caused, resolved, and tested from the technical, engineering, and economic perspective.

First of all, from the technical perspective, practitioners still miss the full landscape of common types of performance issues and their respective resolutions. On the one hand, most existing studies focus on a specific type of performance issues. For example, a well studied type of performance issues is caused by inefficient loop iterations [10], [11], [12], [13], [14]. On the other hand, the few empirical studies [4], [6] that reveal the categorization of common performance issues are based on relatively small datasets—e.g. only 100 and 109 performance issues are investigated in [4] and [6], respectively. They usually focus on a specific programming language or a specific software project domain, such as *JavaScript* [9] or *Android* smart-phone applications [7], [8]. This study aims to contribute a comprehensive overview of different types of performance issues based on a much larger scale dataset containing 570 real-life performance issues. These issues are from open-source projects implemented in three popular programming languages—*Java*, *Python*, *C/C++*. This enables the analysis of the impact of programming language on the causes and resolutions of performance issues in practice.

In addition, in existing literature, performance issues are often treated by code-level fixes, e.g. a few lines of code revision in a single source file [10], [13]. However, practitioners often encounter performance issues with architectural roots, including the unwitting use of architectural performance anti-patterns, such as god classes that induce foci of overload in hardware or software objects [15]. To the best of our knowledge, we are the first to investigate performance optimization from a design perspective. More specifically, we divide performance optimization into two general types: 1) the localized optimization which is implemented in a few lines of code in a single source file; and 2) the design-level

- *Yutong Zhao, Lu Xiao, and Andre B. Bondi are with the School of Systems and Enterprises, Stevens Institute of Technology, Hoboken, NJ 07030 USA. E-mail: {yzhao102, lxiao6, abondi}@stevens.edu.*
- *Bihuan Chen is with the School of Computer Science, Fudan University, Shanghai 200437, China. E-mail: bhchen@fudan.edu.cn.*
- *Yang Liu is with the School of Computer Engineering, Nanyang Technological University, Singapore 639798. E-mail: yangliu@ntu.edu.sg.*

Manuscript received 23 September 2021; revised 6 April 2022; accepted 11 April 2022. Date of publication 14 April 2022; date of current version 13 February 2023.
This work was supported in part by the U.S. National Science Foundation (NSF) under Grants CNS-1823074, CCF-1909763, and CCF-2044888.
(Corresponding author: Lu Xiao.)
Recommended for acceptance by P. Runeson.
Digital Object Identifier no. 10.1109/TSE.2022.3167628

optimization that involves coordinated revision of a group of related source files and their design structure. A typical example of localized optimization is the resolution of fixing inefficient iterations by adding a single line checking condition to "break" the loop. While a typical example of design-level optimization is the resolution of issue *AVRO-753* [16] (discussed in detail in Section 5). The program becomes very slow for special input types. In the resolution, developers employed a factory design pattern to separate and encapsulate the algorithms for treating different input types into different factories. Without the factory design pattern, there will be a potential god class overloaded with responsibilities for treating all input types. In this study, we reveal the typical design-level optimization patterns in resolving performance issues.

Furthermore, from the engineering perspective, software testing is an inseparable part of modern software engineering process, especially for agile approaches [1], [17], [18], [19]. Practitioners often rely on test cases to identify performance issues before releasing the product [2], [19]. Thus, testing should also be an essential aspect in performance optimization. However, there is little understanding regarding whether and how performance optimization leads to revision of related test cases. This study fills this gap by conducting a holistic analysis of the test-production co-change patterns that appear in performance optimization. The goal is to provide insights for practitioners to treat performance optimization and testing as an integrated task.

Finally, software development is constrained by limited time and resources in practice. However, most existing work treats performance issues from purely technical perspectives, without considering the economics behind resolving performance issues. To fill this gap, this study evaluates the ROI (Return on Investment) for addressing each performance issue in our dataset. As the proxy for the investment, we measure the number of engaged developers and the number of discussions for resolving each issue. As for the return, we measure the extent of performance improvement from fixing each issue. Additionally, we argue that performance optimization may also concern other long-term benefits as the return, such as design quality and code maintainability. The tricky part is that these benefits are mostly implicit to measure. Thus, we extract related information from developers' discussions and comments when they address the performance issues. We compare the ROI between the localized and design-level optimization resolution to provide for the purpose of prioritizing different optimization strategies in practice.

In summary, this study fills the above gaps in practitioners' understanding regarding real-life performance issues by answering four research questions:

- *RQ1-What are the common root causes and resolutions of performance issues?* We summarize and categorize common types of performance issues based on 570 real-life performance issues from open source projects implemented in three commonly used languages, Java, Python, and C/C++. We also conduct extensive literature review to find available tools for detecting and fixing related performance issues. We are particularly interested in investigating the impact of programming language on performance issues.
- *RQ2-Whether and how performance issues are addressed by design-level optimization?* We first show how often performance issues in our dataset are resolved by design-level optimization. Next, we reveal the common design-level optimization patterns and why they are necessary in resolving some performance issues. In particular, we are interested in finding out whether the design-level optimization patterns differ for issues in different programming language.
- *RQ3-Whether and how does the testing code change with performance optimization?* We first measure how often developers revise test code in performance optimization. Next, we investigate to what extent the revised test code and production code are related to each other, and what are the test-production co-change patterns in performance optimization.
- *RQ4-What is the ROI for fixing performance issues?* We aim to compare the ROI of localized and design-level optimization. In addition, we examine whether and how the ROI of performance optimization differs for different programming languages.

This work is a substantial extension to our prior work [20], including the four key aspects:

- The original study only includes 192 performance issues from Java projects; while this study extends the dataset to include additional 378 performance issues from Python (162 issues) and C/C++ (216 issues) projects. This allows us to gain a more comprehensive understanding of real-life performance issues.
- We particularly focus on analyzing the impact of programming language on the root causes, the design-level optimization patterns, the ROI in performance optimization.
- We add a new research question that investigates whether and how test code changes in performance optimization. This provides a holistic angle to examine test code maintenance and evolution as an integral engineering process in performance optimization.
- In the ROI analysis, we investigate whether and how other aspects of concerns, such as maintainability, co-occur as return with performance optimization. In particular, we reveal whether localized and design-level optimization associate with other aspects of concern differently.

The key motivation of this extension is to help practitioners gain a comprehensive understanding of real-life performance issues that are based on different programming languages. We chose *Java*, *Python*, and *C++*, because these three programming languages are the most popular object-oriented programming languages in the recent decade [21], [22]. Due to their popularity and wide use in practice, our study can potentially benefit a broad group of practitioners who use these three languages in their daily practice. The comparison of the three programming languages cross-cuts three research questions. The goal is to reveal potential impact of programming language on root causes (RQ1), design-level optimization (RQ2), and ROI (RQ4). This sheds light on the potential impact of programming language on the performance issues for practitioners. It is worth noting

that, overall, the root causes, design-level optimization patterns, and performance improvement factors are not significantly impacted by the choice of programming language. While performance issues in *C++*, tend to yield the highest performance improvement (more than 200 times improvement), compared to *Java* and *Python*.

This study contributes unique insights for practitioners who concern software performance in the following aspects:

- *Root cause*: We summarize **eight** common root causes of real-life performance issues, which are general to the three programming languages and provide a comprehensive overview for practitioners. Developers may benefit from existing tools that focus on different root causes, but the availability and usability of these tools could be an issue.
- *Design-level Optimization*: A non-trivial portion of real-life performance issues requires design-level optimization, despite the programming language. We summarize four types of design-level optimization patterns, including 1) Classic Pattern, 2) Change Propagation, 3) Optimization Clone, and 4) Parallel Optimization. This provides empirical insights for practitioners in terms of how performance issues may be addressed at the design-level.
- *Test/Production Code Co-change*: Only 15% of the performance issues involve test code revision. From these issues, we discover five test-production co-change patterns, including 1) Method Replacement, 2) Performance Input Revision, 3) Test Logic Modification, 4) Test Case Addition, and 5) Test File Addition. Among these five patterns, only the Performance Input Revision pattern directly targets at verifying the performance of the program in different input, while the other four patterns indirectly verify the effectiveness of the performance optimization by focusing on the functional logic of the revised production code. Therefore, performance testing is a potential weak point in the practice of regression testing.
- *ROI*: Generally, design-level optimization requires more investment to implement compared to localized optimization, and does not always result in higher performance improvement. However, for all the three programming languages, design-level optimizations are more likely to be associated with return in other aspects of concerns, such as maintainability and readability, etc. We imply that one motivating scenario for design-level optimization is for achieving the long-term benefits while fixing performance issues.

The remainder of the paper is organized as following. Section 2 introduces the background techniques for us to analyze design-level optimization resolution. Section 3 discusses the research questions and the rationale in detail. Section 4 introduces the dataset and overall study approach for answering the research questions. Section 5 presents the study results. Section 6 discusses the threats and limitations in this work. Section 7 discusses related work. Section 8 concludes the paper.

	1	2	3	4	5
1 TrueTypeFont	(1)				
2 PDTrueTypeFont	ext	(2)			
3 GlyfCompositeDescript	dp		(3)		
4 TTFGlyph2D	dp	dp		(4)	
5 CIDType2Container	dp				(5)

(a) *DSM* Example: PDFBox-2303 (Before Revision)

	1	2	3	4	5	6
1 GlyphTable	(1)	+dp				
2 TrueTypeFont	+dp	(2)				
3 PDTrueTypeFont	+dp	ext	(3)			
4 GlyfCompositeDescript	+dp	-dp		(4)		
5 TTFGlyph2D	+dp	dp	dp		(5)	
6 CIDType2Container		dp				(6)

(b) *Diff-DSM* Example: PDFBox-2303 (After Revision)

Fig. 1. Examples of *DSM* and *Diff-DSM*.

2 Background

2.1 Design Structure Matrix (DSM)

Design Structure Matrix (DSM) is proposed by Baldwin and Clark [23] to capture the complexity of system design. A *DSM* is a square matrix, in which each design variable corresponds both to a row and a column of the matrix. A cell is checked if and only if the design decision corresponding to its row depends on the design decision corresponding to the column. A *DSM* represents modules as blocks along the diagonal.

DSM is often used in modeling software systems [24], [25]. The elements can represent source files. Each cell captures the different types of structural dependencies of the file on the row to the file on the column. In this work, we model two general types of structural dependency: 1) "Ext," which indicates that the file on the row extends the file on the column; and 2) "dp," which indicates other general types of structural dependency, such as method call, from the file on the row to the file on the column. Fig. 1(a) is a *DSM* example containing 5 source files that associate with performance issue *PDFBOX-2303* (elaborated later) in open source project PDFBox. The rows and columns represent the source files, arranged in the same order. The cells display the dependencies among these files. For example, Cell[2, 1] says "ext," meaning *PDTrueTypeFont* (row 2) extends *TrueTypeFont* (row 1). Cell[4,2] says "dp," meaning *TTFGlyph2D* (row 4) has a structural dependency on *PDTrueTypeFont* (row 2).

2.2 Diff-DSM

In this work, we use *Diff-DSM*, which is built upon the *DSM*, to specifically capture the design structure change in software development. *Diff-DSM* has the following uniqueness for the purpose of this study: 1) it only contains files and their structural dependencies involved in a revision; 2) it highlights the added and removed files in the revision; and 3) it highlights the added and removed structural dependencies among involved source files. A *Diff-DSM* is automatically computed by taking a revision ID and the project repository as the input [26].

As an example, Fig. 1(b) is a *Diff-DSM* showing the difference in design structure resulted from fixing the performance issue *PDFBOX-2303*. This issue causes slowness while rendering large external glyphs (e.g. Asian glyphs) such as "STXihei". The reason is that the program has to load and construct objects for 37255 glyph tables and each table is 15.7 Mb. To resolve this issue, developers created a new file, named *GlyphTable*, which is responsible of loading individual glyph table only when necessary. Fig. 1(b) illustrates the overall design structure change for this resolution. First, the top row, *GlyphTable*, is highlighted in green, indicating that *GlyphTable* is a new file added in this revision that contains the core strategy for eliminating unnecessary data loading. Second, Cell[2, 1], Cell[3, 1], Cell[4, 1], and Cell [5, 1] say "+dp" (in green font), which indicates that four files from *TrueTypeFont* (row 2) to *TTFGlyph2D* (row 5), added dependencies on the new file *GlyphTable* (row 1) to leverage the optimized loading strategy. In addition, Cell [4, 2] says "-dp" (red font), meaning *GlyfCompositeDescript* (row 4) removed dependency from the original *TrueTypeFont* (row 2) which contains the original loading method that leads to redundant and unnecessary data loading. As shown in this example, the *Diff-DSM* provides a lens for us to review the rationale of design structure change for performance optimization.

3 Research Questions

As mentioned earlier, this paper aims to answer four research questions. Here, we discuss each RQ in detail.

RQ1: What are the common root causes of real-life software performance issues? Practitioners should be aware of the common types of performance issues to be able to effectively prevent, identify, and fix performance issues. We answer this question in two parts:

- *RQ-1.1 What are the common root causes of performance issues? And, how does the programming language impact the root causes?* The motivation of this sub-RQ is to evaluate the impact of the programming language on the root causes. On the one hand, we aim to reveal whether the root causes are general to all three languages; and on the other hand, we are also interested to see if certain root causes are more relevant in a particular language. Related observations can provide empirical insights for practitioners using different languages.
- *RQ-1.2 How well is each root cause addressed with available tools?* We aim to identify tools from existing literature that are available to detect and fix performance issues in real-life projects implemented with different programming languages.

RQ2: Are performance issues addressed by design-level optimizations, and if so, how? We hypothesize that some performance issues require design-level optimization to maximize performance improvements and ensure code quality at the same time. We address this RQ in two parts as well:

- *RQ-2.1 What percentage of performance issues require design-level optimization? And, what are the typical design-level optimization patterns?* We distinguish local and design-level optimization resolution based on the scope of change in fixing performance issues. For the design-level resolution, we further investigate what are the typical design-level optimization patterns and why they are necessary in addressing some performance issues.
- *RQ-2.2 Does programming language impact the design-level optimization patterns?* We examine whether and how the programming language impacts the design-level optimization patterns and their distribution.
- *RQ-2.3 Do root causes or project domains impact the design-level optimization patterns?* We examine the distribution of the design-level optimization over two other dimensions, including the root causes and project domains. The goal is to reveal whether these factors impact the choice of design-level optimization patterns.

RQ3: Does the test code change with performance optimizations, and if so, how? Software testing is an integrated part of modern software development. Software performance issues are often detected by executing the test cases to find the "hot-spots" [27], [28], [29]. This RQ investigates how likely the test code changes together with the performance optimization, and also reveals the nature of the co-change between test code and production code in performance optimization. We address this RQ in two parts:

- *RQ-3.1 How often do practitioners change test code in performance optimization?* We assume that design-level optimization is more likely to involve test code revision compared to localized optimization due to the larger change scope and complexity of change.
- *RQ-3.2 What are the common test-production co-change patterns in performance optimization?* We reveal whether and how the revision of production code for performance optimization cause changes to the test code. We categorize the test-production co-change patterns to understand the nature of causality.

RQ4: What is the ROI (Return on Investment) for fixing performance issues? Software development is constrained by limited resource and time, and concerns quality attributes other than performance,. This RQ helps practitioners treat performance issues economically. We address this RQ in two parts as well:

- *RQ-4.1 What is the overall ROI for addressing performance issues? In particular, do developers concern about other aspects of benefits in performance optimization?* We measure the number of involved developers and the number of discussions as the proxy of investment, and measure the extent of performance improvement as the return. In addition, we investigate whether developers also concern other aspects of benefits, such as maintainability and readability, when addressing performance issues.
- *RQ-4.2 How is the ROI of localized and design-level optimization compared to each other?* We focus on comparing the ROI of localized and design-level optimization. This provides insights for practitioners to prioritize the two optimization strategies. In particular, we assume that developers prone to design-level optimization when they also concern about other aspects of benefits, such as maintainability, as the return.

TABLE 1
Facts About the Study Subjects and Performance Issues

ID	Subject	*Apache* Category	Description	Focused Lan in Code Base (%)	# Issues	# Perf-key	# Verified	# Resolved	# Issues Merged by PL
1	PDFBox [30]	Content, Library	PDF Library	Java (99%)	3,855	135	93	74	
2	Avro [31]	Big-data, Library	Data Serialization	Java (44%)	2,151	135	113	41	
3	Groovy [32]	Library	Programming Language	Java (98%)	8,476	137	107	36	
4	Collections [33]	Build Management	Dependency Manager	Java (99%)	435	51	46	23	
5	Ivy [34]	Build Management	Dependency Manager	Java (95%)	1,522	54	41	18	192
6	Beam [35]	Big-data	Streaming Data Model	Python (18%)	7,940	254	155	98	
7	Qpid [36]	Server-client	Enterprise Message System	Python (100%)	7,778	102	97	35	
8	LibCloud [37]	Cloud	Storage Manager	Python (99%)	593	29	20	18	
9	Climate [38]	Content	Climate Model	Python (96%)	857	16	13	9	
10	PyLucene [39]	Search	Search Engine Library	Python (39%)	42	4	2	2	162
11	PHP [40]	Language	Programming Language	C++ (97%)	123,660	309	186	149	
12	Kudu [41]	Big-data	Storage Manager	C++ (90%)	2,336	54	38	34	
13	Mesos [42]	Cloud	Cluster Manager	C++ (92%)	7,101	41	38	33	216
			Total		166,747	1,421	939	570	570

Note: Column 1 ("ID") shows the ID of the studied project.
Column 2 ("Subject") shows the name of the project.
*Column 3 ("*Apache *Category") shows the domain of the project, as specified on the* Apache Software Foundation *[43].*
Column 4 ("Description") is a brief description of the project.
Column 5 ("Focused Lan in Code Base (%)") shows the focused programming language and its percentage in code base of the project.
Column 6 ("# Issues") shows the total number of issues in the issue tracking system of the project.
Column 7 ("# Perf-key") shows the number of issues that match the keywords that are relevant to performance.
Column 8 ("# Verified") shows the number of manually verified performance issues of the project.
Column 9 ("# Resolved") shows the number of verified performance issues that have code revision in the project.
Column 10 ("# Issue Merged by PL") shows the number of studied performance issues for each programming language.

4 Study Subjects and Approach

4.1 Study Subjects

This study focus on a total of 570 performance issues from 13 open source projects from the *Apache Software Foundation* [43]. Table 1 summarizes the facts about the study subjects and performance issues.

These subjects were selected based on the following considerations. First, these projects are implemented in three very commonly used programming languages, including *Java* (row 1 to row 5), *Python* (row 6 to row 10), and *C++* (row 11 to row 13). Of a particular note, as shown in the column "Focused Lan in Code Base (%)," for most projects, we study the main programming language, which accounts for more than 90% of the code base. *Avro*, *Beam*, and *PyLucene* are exceptions—the language we focus on takes 44% (*Java*), 18% (*Python*), and 39% (*Python*) of their code bases, respectively. When analyzing each issue, we confirm the programming language in the issue resolution. Thus, whether we focus on the main language of a project does not impact the validity of our findings. Overall, our study includes 192 performance issues from *Java* projects, 162 from *Python* projects, and 216 from *C++* projects. This helps us to investigate the impact of language on the causes and resolutions of real-life performance issues. Second, the projects are in different domains as shown in the column "Description". Thus performance issues from these projects represent diverse features of various domains. The data diversity helps us draw general observations of real-life performance issues of different natures. Third, these projects are all well accepted, successful, and are all still active in the open source community. Lastly, the source code, version control repository, and issue-tracking systems of these subjects are all well organized and readily available on the *Apache Software Foundation* [43]. They provide high quality performance issue data for our study.

4.2 Study Approach

Fig. 2 shows the overview of our approach, which comes in five main steps:

- *Step 1: Data Collection.* We collect the performance issues from the issue tracking systems of the selected projects.
- *Step 2: Root Cause Analysis.* We perform root cause analysis of the retrieved performance issues by inspecting both the issue reports and the code revision, and then we identify available tools for addressing related issues from the literature (RQ1).
- *Step 3: Design-Level Optimization Analysis.* We focus on analyzing the performance issues that require design-level optimization (RQ2).
- *Step 4: Test/Production Code Co-Change Analysis.* We analyze the test code revision involved in performance optimization (RQ3).
- *Step 5: ROI Analysis.* We examine the Return on Investment (ROI) of the performance issues (RQ4).

Table 2 lists the data items that we extract in the study steps for synthesizing the results and insights for the RQs. For the sake of transparency, the detailed data described in Table 2 is shared here: https://doi.org/10.5281/zenodo.6383167. In the following, we elaborate each on step:

4.2.1 Step 1: Data Collection

As shown in Table 1, we initially collected a total of 166,747 issues the issue tracking systems of the 13 projects. These issues dated back to the creation of each project. First, we apply keyword matching to select issues that are relevant to performance, similar to the practice in prior studies [6], [44], [45]. The keywords used include: "*fast, slow, perform, latency, throughput, optimize, speed, heuristic, waste, efficient, unnecessary, redundant, too many times, lot of time, too much time*". These are the combination of the keywords used in previous

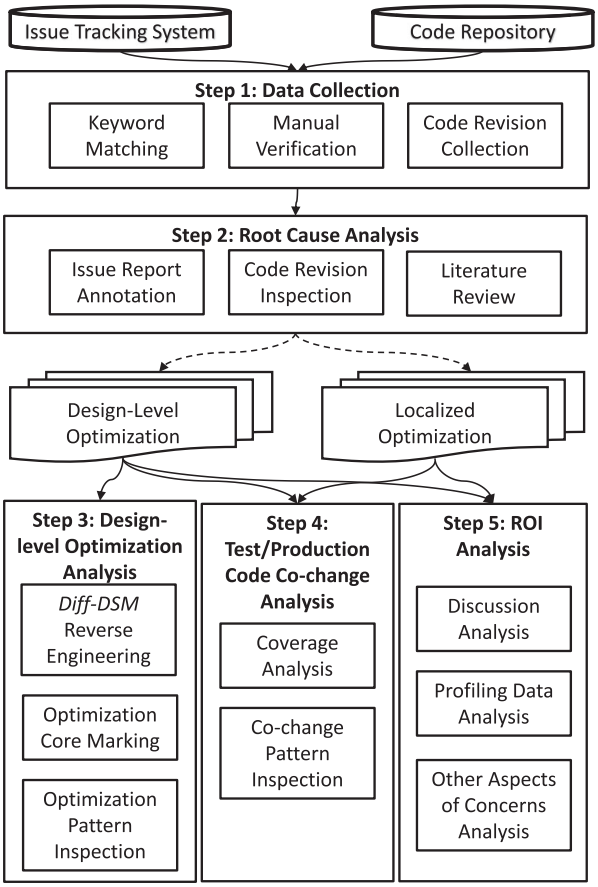


Fig. 2. Study Overview.

studies. If the summary or description of an issue report contains one or more keywords, it is potentially a performance issue. As shown in the fourth column in Table 1, a total of 1,225 issues were kept by matching relevant keywords.

Next, we manually read and verify each issue report to exclude false positives. For example, "performance" sometimes refers to the productivity of the developers. This further distills a total of 868 issues shown in Column "Verified" of Table 1.

We identify and collect the code revision(s) for each performance issue from either 1) the proposed patch(es) in the *Apache JIRA* issue reports if available, or 2) from the projects code repository, such as *Github*, by locating the issue ID that appeared in the commit message. Issues without linked solutions, either because they were not solved or because the linkage was missing, are dropped. We finally identified 570 resolved performance issues for this study. These issues are referred to by their IDs (i.e. D1 in Table 2).

In preparing for the following steps, we manually annotate four key aspects of information (if available) in each issue report. They are: 1) the text that describes root cause (D2), 2) the proposed solution, 3) the profiling data (D12), and 4) any other aspects of concerns (D15) e.g. maintainability, readability, etc. Fig. 3 is an example of an annotated issue report, *PDFBOX-591* [46].

4.2.2 Step 2: Root Cause Analysis

This step aims to reveal the recurring root causes (RQ-1.1) and identify tools from the literature (RQ-1.2).

For RQ-1.1, we apply open coding to derive the root cause categorization [47]. First, we summarize the root cause of each issue in brief terms (D3) based on the extracted root cause description (D2) from the issue report. Also, we manually inspect and summarize the code revision(s) of each issue to gain in-depth understanding of the root cause and resolution to consolidate the summary in D3. We first do so on 50 randomly selected issues of each programming language to generate the initial set of *D3* coding of the root causes. Next, we involve all authors to discuss, merge, and consolidate the *D3* codes to generate the initial code book (D4) that captures the recurring root causes. For example, "*loop break*" and "*infinite iteration*" are merged to *Inefficient Iteration* performance issues. Similarly, "*adding buffer*" and "*cache optimization*" are merged to *Repeated Computation*, since the goal of adding or optimizing a buffer or a cache is to store the calculated results to avoid repeated computation. In the follow-up analysis of the remaining issues, we also summarize the root cause of each issue in brief terms (D3), and then categorize each issue following the D4 root cause code book. The authors of this study hold group discussion for resolving uncertain cases in this process. We found that the initial code book based on 50 issues

TABLE 2
Data Extraction and Synthesis for Analyzing Performance Issues

ID	Data Item	Description	Related RQ
D1	Issue ID	ID of the performance issue.	RQ-1.1
D2	Root Cause Text	Extracted text from issue description that describes to root cause.	
D3	Open Coding	Results of manual open coding process for summarizing root causes.	
D4	Root Cause	Categorized root causes of the performance issue.	
D5	# PC Files	The number of revised production code files.	RQ-2.1
D6	Optimization Level	The optimization level, i.e., localized or design-level, of the performance issue.	
D7	PC Summary	The summary of revised production code.	RQ-2.2
D8	DL Pattern	The categorized optimization pattern of the design-level optimization.	
D9	# TC Files	The number of revised test code files.	RQ-3.1
D10	PC-TC Linkage	The linkage between production code and test code revision.	
D11	TC Summary	The summarized key changes to of the test code.	RQ-3.2
D12	Co-Change Pattern	The categorized test/production code co-change pattern.	
D13	Improvement Factor	The improvement factor of the performance issue after revision.	RQ-4.1
D14	# Developers	The number of involved developers in the performance issue.	
D15	# Discussions	The number of discussions (comments) in the performance issue.	
D16	Other Concerns	The other aspects of concerns while resolving the performance issue.	RQ-4.2

PDFBox-591: PDFBox Performance Issue: BaseParser.readUntilEndStream() rewrite
Root Cause: The current implementation of this method uses a very slow test for end of stream conditions. A profile of the readUntilEndStream() method shows that a huge chunk of the method's processing time is being consumed in the cmpCircularBuffer() call - which is purely part of the test for the end of stream marker. In other words, the readUntilEndOfStream() is spending twice as much time testing for the end of stream marker as it is reading bytes from the stream.
Proposed Solution: A better solution is to use a simpler, direct fail-fast test conditional structure that uses byte primitives. I strongly recommend that the current method be removed and replaced with the following code below.
Profiling Data: This results in a relative speed up of readUntilEndStream() method of a little over a factor of 3 (a ratio of 113/37 = 3.05 if you want to be more precise). This in turn helps the overall performance of PDDocument.parse() by about a factor of 2.7.
Other Concerns: Note the addition of some byte constants used to make the code readable."

Fig. 3. Issue Annotation-PDFBOX-591.

of each language is quite representative of the remaining issues, therefore, no new code (root cause types) are added in the following up analysis. We also found that, some issues match multiple root causes. For example, *PDFBOX-1337* [48] matches both *Inefficient Synchronization* and *Inefficient Data Structure*. In this issue, a dead lock appears with more than 6 threads. Developers replaced the original data structure, *SynchronizedMap*, with a more efficient data structure, *ConcurrentHashMap*, which avoids the multi-thread blocking.

For RQ-1.2, we perform an extensive literature review to identify the tools for detecting and/or fixing performance issues of different root causes. The literature review follows the procedure for performing systematic reviews [49]. The literature selection is composed of three rounds:

- *1st Round—Initial Selection:* we define the search string as "*(Software)* **AND***(Performance)* **AND***(<the name of a root cause>)*," and retrieve the initial set of literature from *Google Scholar* digital library. We manually review the top ranked papers and stop searching when the retrieved papers became irrelevant—based on our experience, this usually stops at the top 30 papers.
- *2nd Round—Snowballing:* We apply the snowballing search [50], [51] to further retrieve more relevant literature that are referenced in papers from the 1st Round.
- *3rd Round—Expansion:* We expand the "name of a root cause" in the search string based on synonymous found in the "snowballed" literature in the 2nd Round. This helps us to retrieve literature that use different names for the same type of performance issues. For example, we found that "*data structure*" is also called "*collection*" or "*container*" in different literature [52], [53], [54], [55].

In each round, we select the most relevant literature by following the inclusion and exclusion criteria listed in Table 3.

TABLE 3
Inclusion and Exclusion Criteria

ID	Inclusion Criteria
I1	The paper is peer-reviewed.
I2	The paper is written in English.
I3	The paper is not grey literature, e.g. technical report, patent, or working in progress.
I4	The paper is published in a international conference, journal or symposium.
ID	**Exclusion Criteria**
E1	A previous version of the paper whose extended version has been included.
E2	The paper is a secondary study (literature review) of existing techniques/ approaches.
E3	The paper does not addresses real-life performance issues.
E4	The paper does not contribute a tool for detecting/fixing performance issues.

There are 42 papers in our final list. We carefully read and annotate related data items in each paper as listed in Table 4. The detailed data is shared with the link https://doi.org/10.5281/zenodo.6383167, following the schema of Table 4. We derive the insights and conclusions for RQ-1.2 based on the targeted root cause (PD4), usage (PD5), programming language (PD6), and link (PD7) to the tools.

4.2.3 Step 3: Design-Level Optimization Analysis

This step aims to identify design-level optimization (RQ-2.1) and reveal the typical patterns (RQ-2.2).

First, for RQ-2.1, we count the number of revised source code files for each performance issue in the project version control system. Admittedly, developers may simultaneously makes a group of changes in one commit, thus revising a group of files does not always imply a design-level optimization. For instance, developers may combine multiple change requests, e.g., fixing a functional bug, with the performance optimization. Thus, we manually verify and exclude the code revision where a group of source files is not revised due to performance optimization. This is done based on our understanding of the code revision (D7) and the description of root causes (D2 and D3) to performance issues. For example, in issue *PDFBOX-1924*, the main purpose is fixing a functional bug. Developers revise four source files, but only one line of code is for improving performance. We record this sanitized number as D5 in Table 2.

Based on the scope of code revision, specified in D5, we distinguish performance issues into two types: 1) *localized optimization* that revises a single production code file; and 2) potential *design-level optimization* that simultaneously revises a group of related production code files. We record the level of optimization as D6 in Table 2 for answering RQ-2.1.

Next, to answer RQ-2.2, we summarize the logic of the code revision (recorded as D7 in in Table 2) and adopt

TABLE 4
Annotated Data Items for Existing Tools

ID	Data Item	Description
PD1	Title	The title of the paper.
PD2	Year	The year when the study is published.
PD3	Tool	The name of proposed tool.
PD4	Root Cause	The root cause of involved performance issues.
PD5	Usage	If the tool can automatically detect and/or fix performance issues.
PD6	Language	The programming languages of performance issues.
PD7	Link	The web link for downloading and installing the tool if available.

Diff-DSM modeling approach [56] to formally and automatically capture which and how source files are involved in the performance optimization. As introduced in Section 2, the *Diff-DSM* helps us to capture the essential design structure change in the code revision.

In our prior work [26], we developed a novel design modeling approach, called *DesignDiff*, that models and visualizes the high-level design differences resulting from every code revision. Given a commit ID and the code base of a software project, our approach automatically interprets and visualizes the high-level design difference and generates the analysis results as *Diff-DSM* matrices. The implementation overview of the *DesignDiff* approach is composed of two main parts: 1) design difference extraction and 2) modeling and visualizing. In the first part, our approach uses Git APIs to revert the code base into two status before and after a given commit. Next, it retrieves the referenced files set before and after revision to comprehensively capture indirect design impacts of the given commit. In the second part, our approach models and visualizes a commit as a sequence of design changes. It first uses a third-party tool, named *SciTool Understand*, to calculate two graphs of the given commit before and after revision. In the two graphs, the nodes represent the involved code files and the edges represent the dependencies among these code files. Next, a simple graph comparator identifies the differences between the two graphs, including the added or removed code files and modified dependencies. The final output is a *Diff-DSM* that highlights the added or removed files and modified dependencies.

Marking the "optimization core" in each *Diff-DSM* relies on matching the root cause (D4) and source file(s) with the most directly relevant revisions. This matching is based on our understanding of the code revision (D7). In addition to the source code revision itself, usually, the commit messages and the code comments left by developers are very helpful. We also carefully examine the other files that are revised with the "optimization core". This helps us understand how the other files change with the core and provides additional confirmation of the "optimization core".

Fig. 1(b) (discussed in Section 2) contains an example of "optimization core". The root cause of *PDFBOX-2303* [57] is that "*FontBox's TTF GlyphTable reads the entire glyph table and constructs an object for each glyph*". The resolution is to "*modify GlyphTable to make it lazy so that glyphs are read individually only when needed*". Based on our methodology described above, *GlyphTable* is marked as the "optimization core" of the *PDFBOX-2303*, based on the combination of three aspects of information: 1) developers left a comment in it, saying "*we don't actually read the table yet because it can contain tens of thousands of glyphs*"; 2) the revised code in *GlyphTable* indeed implements the lazy glyph loading that avoids unnecessary reading of the entire glyph table; and 3) the other source files are revised to accommodate the changes made to the "optimization core," i.e., "*GlyphTable*".

Finally, we reveal typical design-level optimization patterns (D8), combining our understanding of the code revision logic (D7) and the analysis of individual *Diff-DSM*. The patterns are categorized based on the nature of the relationship between the "optimization core(s)" and other source files in the *Diff-DSM* view. The example shown in Fig. 1(b) is recognized as the *Change Propagation*, since the added dependencies are change prorogation from the optimization core. As we will show in the answer to RQ2, we also observed three other design-level optimization patterns, including *Classic Design Pattern*, *Optimization Clone*, and *Parallel Optimization*.

4.2.4 Step 4: Test-Production Co-Change Analysis

This step aims to reveal test and production code co-change patterns in performance issue resolution.

For RQ-3.1, we count the number of test files (D9) revised with the performance optimization. When counting, we confirm that the test code revision indeed is caused by, or at least closely related to, the revision of the production code under-test for performance optimization. This relies on two aspects of information: 1) the revision logic in the test code and production code; and 2) the revised test case name and the production method name. For localized optimization, this is straight forward since only one production source file is involved. For the design-level optimization, the match focuses on the "optimization core" since it carries the essence of the optimization. The test case name and the product method name provides additional information. It is a common convention that the test case name should imply which production method it is testing. For example, in *PDFBOX-2126* [58], the test case *testMultiplication()* is revised together with production function *Matrix.multiply (Matrix m1, Matrix m2)*, and their names suggest their connection. We record how the test code revision is related to the production code revision as D10 in Table 2.

After confirming the linkage between test and production code revision, we further elaborate and analyze the nature of their co-change (RQ-3.2). For example, in *PDFBOX-2126* [58], the method *Matrix.copy()* is replaced with *Matrix.clone()* in both of the production source code and the test case. In *COLLECTIONS-450* [59], the developers create new test cases to test the new production methods they create for performance optimization. While in *AVRO-1455* [60], the developers change the test case logic according to the production code revision. When analyzing the test case, we found that revision to the input parameters, such as the input array size or thread waiting time, is a unique pattern in test code revision for performance testing. For example, in *AVRO-1090* [61], developers increased the input array size by 25 times (as shown in row 2 and 3 in Fig. 12(b)) to test the efficiency of revised method in production code for clearing a very large array. We record the nature of co-changes as D11 in in Table 2. Finally, we summarize recurring test and production code co-change patterns (D12) based on the cases we review.

4.2.5 Step 5: ROI Analysis

This step focuses on RQ4—The ROI analysis.

First, we estimate the investment for implementing performance optimization based on the number of involved developers and the number of discussions:

- #*Developers (D14)*: The number of developers who participated in the discussion of an issue report. Generally, the more developers involved, the more difficult/expensive to address.

- #*Discussions (D15)*: The number of discussion comments associated with an issue report. More discussions are needed for addressing an issue, it is more difficult/expensive to address.

The number of involved developers (D14) and the number of discussions (D15) are directly crawled from the issue tracking system. We create a program to download the issue report, including description and discussion by the developers, in the format of *XML* files. Then, we use an *XML* parsing program to extract the number of developers (D14) and the number of discussions (D15) submitted by the involved developers from the *XML* files.

Next, for the estimation of return, we consider two distinctive aspects:

- *Improvement Factor (D13):* The extent of performance improvement, as 1) $\frac{ResponseTime\_BeforeFix}{ResponseTime\_AfterFix}$ if performance measured by response time; or 2) $\frac{Throughput\_AfterFix}{Throughput\_BeforeFix}$ if performance measured by throughput. Note that, the related metrics are extracted from the profiling data embedded in issue reports and discussions. We did not collect the profiling data ourselves due to the lack of a systematic and reliable approach to replicate the performance issues.
- *Other aspects of concerns (such as code readability and maintainability):* We annotate and collect the other aspects of concerns (D16) from the issue report descriptions and discussions from *Apache JIRA* issue tracking system. For example, in issue *PDFBOX-591* [46], we extract the related description as "*Note the addition of some byte constants used to make the code more readable*." Thus, we marked *PDFBOX-591* also improved readability while optimizing performance.

The insights and conclusions of RQ-4.1 is derived by comparing the return (D13) and investment (D14, D15) for the performance issues in different level of optimization (D6). RQ-4.2 is based on the other aspects of concerns (D16).

5 Study Results

This section answers the four research questions.

5.1 Root Causes and Resolutions

RQ-1.1 What are the common root causes of performance issues? And, how does the programming language impact the root causes? We observe **eight** general types of root causes that recur in the 570 performance issues. Each root cause has a corresponding typical resolution. We will explain each in the following:

Inefficient Data Structure (IDS): The choice of an inefficient data structure consumes a large amount of memory and/or takes a long time. Typical resolution is replacing the inefficient data structure by a more efficient data structure. Although each of the three programming languages has its specific data structures, the sub-optimal choice of data structure could occur with any of them. For example, in issue *PDFBOX-410* [62] (*Java*), developers replaced *StringBuffer* with *StringBuilder* to improve the efficiency of text loading; in issue *LIBCLOUD-254* [63] (*Python*), developers replaced *LazyList* data structure by a simple iteration; in issue *MESOS-2126* [64] (*C++*), developers replace the *Queue* by *Vector*, since Vector does dynamic allocation only when necessary to reduce CPU cache usage in most cases. This indicates that regardless of the programming language, using the right data structure is important. Table 5 lists the common data structure replacement patterns we observe. The most common case is to replace *Array* or *List* by *Set* or *Map*, which makes data searching faster. *On a particular note, based on our dataset, Java projects are more likely to suffer from this problem.*

TABLE 5
Inefficient Data Structure and Replacement

Inefficient Data Structure	Replacement	# Issues
Array, List	Set, Map	23
HashMap, WeakHashMap	ConcurrentHashMap	8
String, StringBuffer	StringBuilder	7
Integer, Float, Double	int, float, double	4
HashMap	TreeMap	3
LinkedList	ArrayList	2
Others		7
Total		54

Repeated Computation (RC): A program repeatedly performs the same computation and produces the same output because the state from which the output is derived has not changed. Typical resolution is to 1) store the output in a cache or a buffer for re-using [65]; and 2) only perform the computation when the program status changes.

Inefficient Iteration (II): The status of loop iterations remains the same and the iterations become useless. A typical resolution is to check whether the loop status becomes stable; if so, break and exit the loop.

Inefficient API Usage (IAU): Many different APIs provide the same or similar functionalities, but some APIs are more efficient than the others in certain context. This type of problem is caused by sub-optimal choice of APIs [66]. Typical resolution is to replace it with a more efficient one [67], [68], [69].

Inefficient Synchronization (IS): These performance problems are caused by the synchronization issues among multiple threads. It usually happens because different threads have to access the same resource, and thus they have to wait for each other. In the worst cases, threads may even get blocked, resulting in lengthy execution/waiting time. The resolution is to improve the synchronization mechanism.

Redundant Data Processing (RDP): These performance issues are caused by redundant or tedious data processing. For example, a typical scenario is processing (e.g., copying or initializing) a large chunk of data in small units, such as bit by bit, or pixel by pixel (in graph transformation). The typical resolution is to use more efficient data processing strategy to avoid heavy iterative data processing, such as processing all the data in one go, or to eliminate unnecessary data processing.

Inefficiency under Special Cases (ISC): The program runs well most of the time, but it becomes extremely slow or causes memory bloat in special cases [6], [70], [71], [72], [73]. In particular, the inefficiency tends to happen when the input is either null or super large. Typical resolution is to 1) add checking conditions for the special cases, and 2) employ special algorithms to treat each special case efficiently.

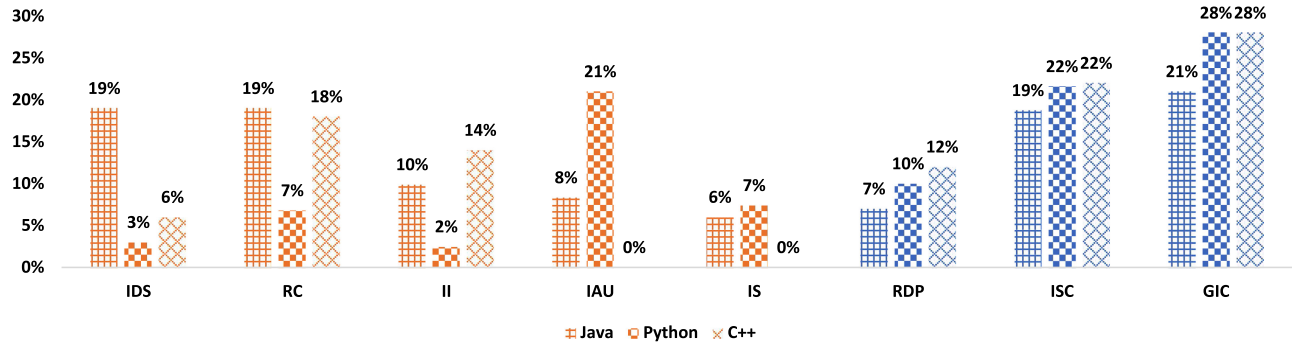


Fig. 4. Percentage of Root Causes in Different Languages.

General Inefficient Computation (GIC): These performance issues are caused by other general inefficient in the algorithm. They are usually addressed by specific algorithmic improvements. As an example, in issue *PDFBOX-600* [74], the order of two checking conditions in an *and* operator caused inefficiency, since in most cases, the first checking condition is true and the second checking condition is false. The developers switched the order to avoid checking both conditions in most cases.

Impact of Programming Language on Root Causes: Fig. 4 shows the percentage of performance issues associated with the *eight* root causes. We found that the proportions of three root causes, *Repeated Computation (RDP)*, *Inefficient Special Case (ISC)*, and *General Inefficient Computation (GIC)* are quite consistent for the three languages. In comparison, the other root causes are more likely to be associated with a certain programming language. For example, *Java* projects (19%) are more likely to have *Inefficient Data Structure (IDS)*, compared to *Python* (3%) and C++ (6%) projects. *Inefficient API Usage (IAU)* is most common in Python (21%), and it does not appear in C++ projects. This could be because Python programming relies heavily on the choice of existing APIs; while C++ projects tend to heavily rely on the low level function implementation. However, we acknowledge that this could be caused by our dataset, and may not be generalized by another dataset.

We leverage the *Mann Whitney U Test* to provide some quantitatively insights. We check the *p-value*, which indicates the level of significant difference between the two groups of samples. According to [75], p-value < 0.05 (5%) or even p-value < 0.01 (1%) indicates significant difference. As shown in Table 6, programming language does not have statistically significant impact on the root causes.

Of a particular note, the eight types of causes are not mutually-exclusive. Some issues are associated with multiple root causes. For example, in *PDFBOX-604* [76], developers resolved 1) *Inefficient Data Structure*, by replacing *StringBuffer* with *StringBuilder* to speed up the text encoding process, 2) *Repeated Computation* by allocating a new buffer that memorizes the font type information, and 3) *General Inefficient Computation* by avoiding unnecessary look ups for font sizes, to address the performance issue in "text extraction" in PDFBox from different aspects. Therefore, the percentages of the eight categories do not add up to 100%.

TABLE 6
Impact of Programming Language on Root Causes

Comparison Group	*Java* vs. *Python*	*Java* vs. C++	*Python* vs. C++
All Root Causes	0.87	0.75	0.96

RQ-1.1 Implication: We observe eight common recurring root causes of real-life performance issues. These root causes apply generally to the three programming languages. Practitioners should be aware of these common root causes. This awareness helps practitioners to prevent, identify and resolve performance issues.

RQ-1.2 How well is each root cause addressed with available tools? In summary, we found 42 tools for detecting and/or fixing real-life performance issues from a total of 504 papers searched in three rounds. In the 1st round, we reviewed 240 papers and identified 19 available tools. In the 2nd round, we performed snowballing expansion base on these 19 studies, and identified 14 available tools from 154 papers. Finally, in the 3 rd round, we expanded the search string based on synonymous found in the "snowballed" literature in the 2nd Round, and identified 9 available tools from the retrieved 110 papers. These tools were published between 2003 to 2021 in ICSE (21%), ISSTA (12%), PLDI (12%), FSE (5%), ASE (5%), OOPSLA (5%), and ICPE (5%), and other conferences or journals (36%).

Table 7 lists these tools and the respective root causes that they can detect and/or fix. The first column shows the respective root cause. The second column lists the tool name, with "D" indicating detecting and "F" indicating fixing the performance issues. The third column shows the programming language that the tool addresses. The last column shows the publication year and whether a link to download the tool is available ("A" means available).

Fig. 5 summarizes an accumulative number of tools for the three programming languages over the years. We observe that 1) Overall, there are more tools for *Java* projects (21 tools), compared to C/C++ (15 tools) and *Python* (8 tools); 2) Before 2013, C/C++ has the largest number of tools; and 3) Since 2015, there has been an increasing number of *Python* tools. Next, we will briefly discuss the tools for identifying and resolving each root cause:

1) Inefficient Data Structure (IDS): All the seven tools can monitor the dynamic execution of a program to recommend potential replacements of data structures. Chameleon [53] and Perflint [77] instrument *Java* and C++ applications, respectively, to collect dynamic profiling data, such as

TABLE 7
Existing Tools for Detecting/Fixing Performance Issues

Root Cause	Tool	Language	Year(A)
Inefficient Data Structure	[D,F]: CHAMELEON [53]	Java	2009
	[D,F]:Perflint [77]	C++	2009(A)
	[D,F]: CFL [55]	Java	2010
	[D,F]: Brainy [78]	C++	2011
	[D,F]: ContainerBloat [79]	Java	2012
	[D,F] CoCo [54]	Java	2013
	[D,F]: CollectionSwitch [52]	Java	2018
Repeated Computation	[D,F]: Likwid [80]	C++	2010
	[D]: Cachetor [81]	Java	2013(A)
	[D,F]: MemoizeIt [65]	Java	2015(A)
	[D]: RedSpy [82]	C++	2017
	[D]: LoadSpy [83]	Java	2019
Inefficient Iteration	[D] PET [84]	Java	2010(A)
	[D]: pyCPA [85]	Python	2012
	[D]: Toddler [11]	Java	2013(A)
	[D,F]: Caramel [10]	Java/C/C++	2015
	[F]: Clarity [14]	Java	2015
	[D]: GLIDER [12]	Java	2016(A)
	[D]: LDoctor [13]	Java/C/C++	2017
Inefficient API Usage	[D,F]: Pynamic [86]	Python	2007(A)
	[D,F]: SEEDS [87]	Java	2014
	[D,F]: BIKER [67]	Java	2018
Inefficient Synchronization	[D]: LIME [88]	C++	2011
	[D,F]: SHERIFF [89]	C++	2011(A)
	[D]: PRADATOR [90]	C++	2014
	[D]: SpeedGun [44]	Java	2014(A)
	[D]: SyncProf [91]	C/C++	2016
	[D]: SyncPerf [92]	C/C++	2017
	[D,F]: Pymoo [93]	Python	2020(A)
	[D]: Score-P [94]	Python	2021(A)
Redundant Data Processing	[D]: MRNet [95]	C++	2003
	[D,F]: Simplifier [96]	Java	2012(A)
Inefficiency under Special Cases	[D]: GA-Prof [70]	Java	2015
	[D]: PerfPlotter [97]	Java	2016(A)
	[D]: FOREPOST [98]	Java	2016
	[D]: PerfFuzz [71]	C	2018
	[D]: PerfCI [99]	Python	2020(A)
General Inefficient Computation	[D]: Trend Profiler [27]	C	2007
	[D]: Spectroscope [28]	Perl/C++	2011(A)
	[D]: Perprof-Py [100]	Python	2016
	[D]: PVLIB [101]	Python	2016
	[D]: VyPR [102]	Python	2020(A)

Note: "D" means the tool can automatically detect.
"F" means the tool can automatically provide fixing resolutions.
"A" means the tool has available web link for downloading and installing.

function execution time and number of function calls. These profiling data are then compared with a set of "detection rules" to determine if the data structures should be changed. Brainy [78] is more advanced than Perflint—it feeds dynamic profiling data into a machine learning model, and the model selects the best data structure for performance optimization. Similarly, another *Java*-based tool, *CFL* [55] leverages both the dynamic and static analysis to find inappropriate use of data structures in *Java* programs. ContainerBloat [79] focuses on data structure replacement for reducing memory usage; while Coco [54] focuses on improving the speed. In comparision, CollectionSwitch [52] balances the speed and memory in data structure replacement. Furthermore, CollectionSwitch [52] is more advanced among others, since it does not only support the replacement of simple data structures, such as *Set* and *Map*, but also advanced data structures, such as *HashArrayList*, *TreeMap*, and *CompactHashMap* etc.

2) Repeated Computation (RC): We found three tools, *Likwid* [80], Cachetor [81], and MemoizeIt [65], that can perform dynamic profiling to report operations that repeatedly generating identical data values. Two detecting tools,

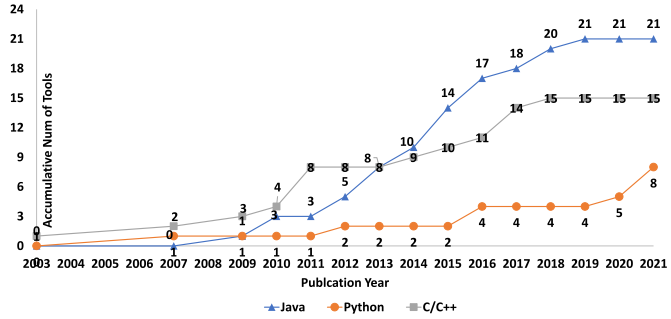


Fig. 5. Cumulative Number of Tools vs. Publication Year.

LoadSpy [83] and RedSpy [82], focus on *Java* and C++ applications, respectively. They can also identify repeated memory loads of the same value or result-equivalent computation which waste both memory and processor functional units. Locating these operations can help developers to add cache or buffer. MemoizeIt [65] is more advanced compared to other tools, because it automatically provides suggestions regarding how to implement the caching mechanism.

3) Inefficient Iteration (II): We found seven tools that can detect *Inefficient Iterations* and add conditional breaks to fix these issues [9], [10], [11], [12], [13], [84], [85]. Each tool relies on a specific resolution for detecting and fixing the *Inefficient Iteration* issues. For example, PET [84] is an automatic *Java* test case generator for exposing iterations that spent large amount of time. Caramel is a static analysis tool that detects inefficient iterations and add conditional-breaks to fix the issue [10]. pyCPA [85] and Clarity [14] are static analysis tools that detects nested loop traversals based on path analysis. Toddler [11] detects inefficient iterations by finding repetitive memory accesses in dynamic execution. However, the effectiveness of Toddler depends on the quality of input tests. Another dynamic tool, Glider, automatically generates tests for exposing unnecessary traversal of iterations [12]. Static analysis tools can only find a subset of inefficient iterations; while dynamic analysis tools are more comprehensive but slowdown the program. Song *et al.* [13] proposed a static-dynamic hybrid analysis tool, LDoctor, which is faster than Toddler and more effective than Caramel.

4) Inefficient API Usage (IAU): We found three tools. Pynamic [86], SEEDS [87] and BIKER [67], for detecting and fixing *Inefficient API Usage* issues. The *Python*-based tool, Pynamic [86], focuses on a specific type of *Inefficient API Usage* that leads to long execution time under heavy file I/O load. *Python* programs usually rely on third-party APIs for reading and writing file from hard disk. But, these APIs may not be efficient. Pynamic [86] wraps hard disk file loading and manipulating utility functions as modules, which replace inefficient APIs. SEEDS [87] focuses on the energy consumption of applications running on battery-powered mobile devices. It automatically detects *Java* APIs that consumes a large amount of energy, and recommends alternative APIs. BIKER [67] is another *Java*-based API recommendation tool that leverages Stack Overflow posts to recommend and prioritize candidate APIs to save execution time and memory usage.

5) Inefficient Synchronization (IS): We found eight tools for detecting and/or fixing *Inefficient Synchronization* issues. They focus on different situations that cause *Inefficient Synchronization*. LIME [88] analyzes parallel programs and

reports load imbalance that causes performance issues. Liu *et al.* focuses on multi-threaded false sharing, which occurs when two threads simultaneously update logically-distinct objects. This may result in invalid data access, and when this happens, new threads will be created to re-access the data. Unnecessary thread creation degrades performance by an order of magnitude [89], [90]. Thus, Liu *et al.* proposed two *C/C++*-based tools, SHERIFF [89] and PRADATOR [90], that can perform per-thread memory isolation to accurately detect false sharing. PRADATOR [90] detects false sharing more accurately than SHERIFF [89], but SHERIFF can also provide optimization suggestions. SpeedGun [44] is an automatic performance regression testing tool for monitoring thread-safe classes in *Java* programs. *Java* thread-safe classes often struggled with two opposite goals: 1) the safety goal requires synchronizing concurrent accesses, and 2) the performance goal requires preventing unnecessary synchronization [44]. The key idea of SpeedGun [44] is to automatically generate multi-threaded performance tests and notify developers of the performance changes of the thread-safe classes after code revision. Similarly, SyncProf [91] and SyncPerf [92] generate multi-threaded performance tests for *C/C++* applications to detect and optimize synchronization bottlenecks. SyncPerf [92] is more accurate than SyncProf [91] in detecting synchronization bottlenecks. For the *Python* based tools, Pymoo [93] evaluates the synchronization of multi-threaded execution in distributed computing. It visualizes the performance profiling data and makes optimization decisions such as modifying the number of parallel threads. Score-P [94] focuses on evaluating the efficiency of thread parallel, process parallel, and accelerator-supported workloads.

6) Redundant Data Processing (RDP): We found two tools for detecting and fixing *Redundant Data Processing* [95], [96]. MRNet [95] monitors data communication between front-end and back-end modules in *C++*-based applications. It finds data transfer that happens in small pieces rather than large bulks. Small pieces of data transfer can cause significant latency, because the synchronization overhead with small pieces of data transfer is much higher than that with large chunks of data transfer. Thus MRNet alerts practitioners once it monitors a large amount of small piece data transfer. Simplifier [96], a *Java*-based tool, eliminates redundant data transformation. It integrates small-sized data fragments (less than 1Kb) into large bulks, and it also eliminate duplicated data.

7) Inefficiency under Special Cases (ISC): We found five tools for detecting performance issues that are caused by *Inefficiency under Special Cases*. Shen *et al.* proposed a *Java*-based tool, named GA-Prof [70], that uses a genetic algorithm to explore a large space of input value combinations for automatically and accurately detecting performance bottlenecks that appear under special combinations of input values. Chen *et al.* proposed another *Java*-based tool, PerfPlotter [97], that captures input probability distribution over execution time for the program. PerfPlotter [97] heuristically explores high-probability and low-probability paths through probabilistic symbolic execution. Once a path is explored, this tool generates and executes a set of test inputs to detect the special inputs that cause slowness of the path. *FOREPOST* [98] is a *Java*-based tools that can also generate a wide range of test inputs, and it focuses on identifying the specific test input that can potentially cause bottlenecks. The *C*-based tool, PerfFuzz [71], can also generate a variety of input values to locate the hot spots that are triggered by a special set of inputs. PerfFuzz is more advanced than GA-Prof because it not only detects performance bottlenecks in a single function, but also provides pathological analysis of the total execution time of a program under different test inputs. The *Python*-based tool chain, PerfCI [99], allows developers to differentiate performance bottlenecks into two types, e.g., that manifests in all inputs ("always-active") and that is triggered on a special condition ("hard-to-detect"). For the latter, PerfCI provides a unit test runner which can execute the application code many times based on different inputs to reproduce the performance bottleneck under a special condition.

8) General Inefficient Computation (GIC): The algorithm design of a program have the most fundamental influence on software performance [27]. Poor algorithm design can lead to inefficient computation in general. The challenge is that the complexity derived from the mathematical analysis of the algorithm design cannot precisely reflect the run-time complexity [29]. Thus, related tools focus on profiling the dynamic execution of programs to identify "hot-spots" that are associated with inefficiency. Goldsmith *et al.* proposed Trend Profiler to measure the run-time complexity by executing a program on workloads spanning several orders of magnitude [27]. Similarly, Spectroscope [28], Perprof-Py [100], PVLIB [101], and VyPR [102], also uses dynamic profiling to detect hot-spots in running programs by collecting data such as the CPU time, number of functions executions, and number of iterations, etc.

> *RQ-1.2 Implication:* Practitioners may benefit from existing tools when facing similar performance issues. However, there are several potential concerns: 1) The applicability of the tools could be a potential problem. The tools have not been tested and compared to each other on any benchmark dataset; and 2) The availability and usability of these tools are potential obstacles for practitioners to using them, given that only 40% tools have available web links and many are no longer actively maintained.

5.2 Design-Level Optimization

RQ-2.1 What percentage of performance issues require design-level optimization? And, what are the typical design-level optimization patterns? Among the total 570 performance issues, the majority (73%) of performance issues were fixed by localized code revisions, while the remaining 27% performance issues were addressed by design-level optimizations. We reveal four typical design-level optimization patterns with the help of the *Diff-DSM* modeling introduced in Section 2:

1) Classic Design Patterns: The developers employ classical design patterns for addressing the performance issues and achieving good design at the same time. For example, issue *AVRO-753* [16] is caused by *Inefficiency under Special Cases*. The *BinaryEncoder* is slow when processing data chunks smaller than 128 bytes. The factory pattern provides an elegant design for treating different input cases

	1	2	3	4	5	6	7	8	9	10
1 BlockingBinaryEncoder	(1)	-Ext, -dp	+Ext, +dp							
2 BinaryEncoder		(2)								
3 BufferedBinaryEncoder		+Ext	(3)							
4 DirectBinaryEncoder		+Ext		(4)						
5 EncoderFactory	+dp	+dp	+dp	+dp	(5)					
6 RpcSendTool				+dp	(6)					
7 RpcReceiveTool				+dp		(7)				
8 BinaryFragmentToJsonTool				+dp			(8)			
9 DataFileReadTool				+dp				(9)		
10 JsonToBinaryFragmentTool		-dp		+dp						(10)

(a) Classic Design Pattern: Avro-753

	1	2	3	4
1 Matrix(*)	(1)			
2 TextPosition	dp	(2)		
3 PDFStreamEngine	dp	dp	(3)	
4 ShowTextEngine	dp			(4)

(b) Type 1 Propagation: PDFBox-893

	1	2	3
1 NumberFormatUtil	(1)		
2 PDFAbstractContentStream(*)	+dp	(2)	
3 PDFContentStream	dp	dp	(3)

(c) Type 2 Propagation: PDFBox-3421

	1	2	3	4	5	6	7
1 PDTrueTypeFont	(1)						
2 PDType1CFont		(2)					
3 PDType0Font			(3)				
4 PDType1Font				(4)			
5 PDType3Font					(5)		
6 PDCIDFontType0			dp			(6)	
7 PDCIDFontType2			dp				(7)

(d) Optimization Clone: PDFBox-3224

	1	2	3	4	5
1 PDFont	(1)				
2 PDSimpleFont	Ext	(2)			
3 COSNumber			(3)		
4 ICU4JImpl				(4)	
5 PDFStreamEngine					(5)

(e) Parallel Optimization· PDFBox-604

Note: "Ext": child class extends a parent class.
"dp": a general dependency except extend or implement.
"-" means the following dependency is removed.
"+" means the following dependency is added.
Files with shaded background are newly added.

Fig. 6. Design-level Optimization Patterns.

in separate. Fig. 6(a) shows the *D-DSM* of this optimization. The developers added three new source files (row 3 to 5), which form a factory design pattern. They are: 1) *BufferedBinaryEncoder* (row 3), a concrete encoder that efficiently deals with large data chunk by using a buffer; and 2) *DirectBinaryEncoder* (row 4), the other type of encoder that efficiently deals with small data chunk without buffer; and 3) *EncoderFactory* (row 5), which is the factory pattern interface;. The *EncoderFactory* is in charge of picking the right encoder with respect to the input size. Thus *EncoderFactory* depends on a bunch of encoders (row 1 to row 4), including the newly added two. Meanwhile, the clients of *Encoder*, such as the tool classes (row 6 to 10) are all changed to refer to *EncoderFactory* to benefit from the proper encoder.

2) Change Propagation: The root cause of a performance issue is addressed in one source file, namely the optimization core; and the optimization core propagates changes to a group of source files that structurally connect to it. There are two types of propagations: Type 1: The optimization core propagates changes to a group of source files that structurally depend on and benefit from the core. For example, Fig. 6(b) is for issue *PDFBOX-893* [103]. The optimization core is class *Matrix* (row 1), which contains *Repeated Computation* of matrix production. It propagates changes to files on row 2 to row 4, which call the core. Type 2: The optimization core propagates changes to a group of source files that the core depends on, to support the core. For example, Fig. 6(c) is a Type 2 propagation for issue *PDFBOX-3421* [104]. The optimization core is

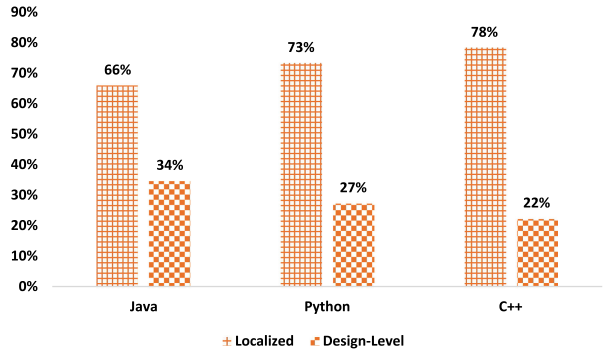


Fig. 7. Localized vs. Design-Level Optimization.

PDAbstractContentStream, which suffers from an inefficient special case. The developers created a new utility class, named *NumberFormatUtil*. When applicable, it is used by the optimization core.

3) Optimization Clone: The developers fix multiple instances of the same performance root cause that are cloned in multiple locations in the code base. We noticed that the involved source files are usually structurally independent from each other. Issue *PDFBOX-3224* [105] is such an example, shown in Fig. 6(d). All the classes in this change is a certain type of *Font*, such as *PDType1Font*. A method, named *getBoundingBox()*, which suffers from repeated computation, is cloned in 7 Font related classes. Therefore, the optimization is also cloned in 7 locations.

4) Parallel Optimization: The developers made parallel optimizations in multiple locations that suffer from different root causes for resolving an issue. In issue *PDFBOX-604* [76] as shown in Fig. 6(e), the developers made five parallel optimization. For example, in *PDFont* (row 1), developers added a cache to memorize font type to avoid repeated computation. In *PDSimpleFont* (row 2), the developers eliminated repeated computation. Each source file here contains a separate optimization, but all belong to the "text extraction" component.

RQ-2.1 Implication: According to Smith and Williams [15], most performance issues have their roots in poor architectural decisions made before coding is done. Our results on these four patterns reinforce this argument, which represent four design strategies to resolve performance issues. Practitioners should be aware of the design-level optimization patterns and make informed design decision when fixing performance issues.

RQ-2.2 Does programming language impact the design-level optimization patterns? As shown in Fig. 7, regardless of the programming language, a non-trivial portion (22% to 34%) of performance issues require design-level optimization.

In addition, as shown in Fig. 8, for all of the three programming languages, the majority of design-level optimization pattern is *Change Propagation*. Furthermore, we perform the *Mann Whitney U Test* to quantitatively evaluate whether the programming language has an impact on the design-level optimization patterns. The *p-values* for the three groups of comparisons, e.g. *Java* vs. *Python*, *Java* vs. *C++*, and *Python* vs. *C++*, are 0.6, 0.75, and 0.76, respectively.

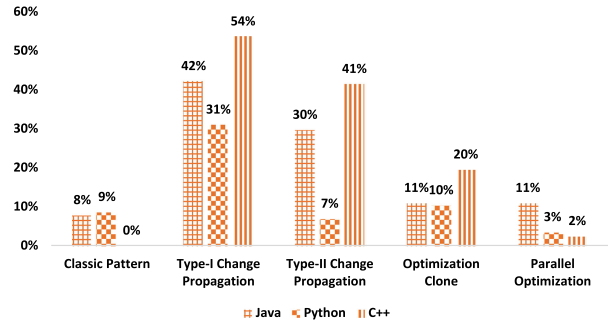


Fig. 8. Design-Level Optimization Pattern Distribution vs. Programming Language.

Thus, we conclude that the four design-level optimization patterns are not impacted by the programming language.

RQ-2.2 Implication: The four design-level optimization patterns generally apply to the three different programming languages. In particular, *Change Propagation* is the most common pattern in all three languages. Developers should pay attention to the "Ripple-effect" in order to correctly and effectively implement performance optimization.

RQ-2.3 Do root causes or project domains impact the design-level optimization patterns?

First, for subject domains, the 13 open source software projects are in seven different domains, as listed in the column "*Apache* Category" in Table 1. This information is specified by the *Apache Software Foundation* [43]. Fig. 9 shows the distribution of design-level optimization patterns over the seven project domains. Overall, we do not observe distinction with the different domains, and Change Propagation is the most common pattern for all. Therefore, we conclude that domain should not have much impact on the design-level optimization pattern.

Second, Fig. 10 shows the distribution of design-level optimization patterns over different root causes. The most unique root cause is *Inefficient Iteration (II)*, which us exclusively associated with the *Optimization Clone* pattern. It is because the resolution of *Inefficient Iteration (II)* issues is to simply add conditional breaks. Related design-level issues are adding breaks in multiple locations, which leads to *Optimization Clone*. Otherwise, we did not observe much impact on the choice of design-level optimization patterns in the other root causes.

RQ-2.3 Implication: The design-level optimization patterns are general to different subject domains and root causes.

5.3 Test Code Changes

RQ-3.1: How often do practitioners change test code in performance optimization? Among the 570 performance issues, only a small portion (15%) involve test code revisions. Specifically, there are 27%, 13%, and 6% involve test code revision in *Java*, *Python*, and *C++* performance issues, respectively. Furthermore, among performance issues that involve test code revision, the majority, 75%, only revise 1 test file; 13% revise 2 test code files; and 12% revise 3 or more test files.

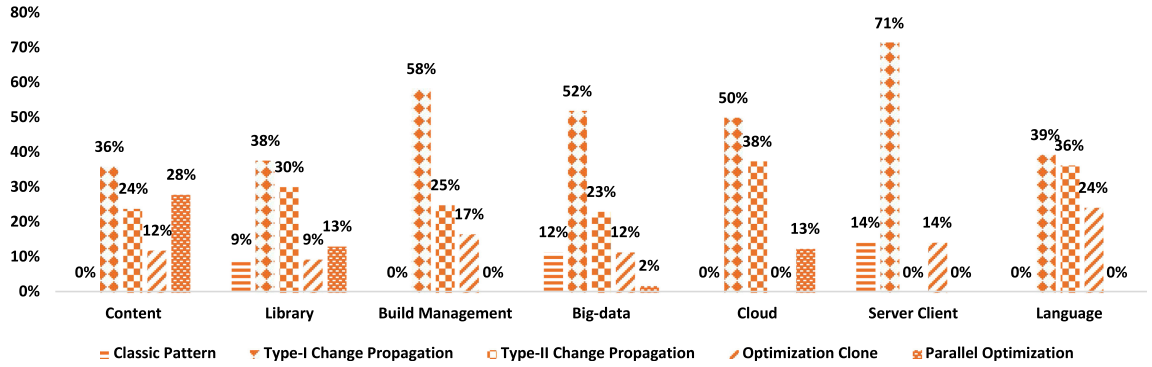


Fig. 9. Design-Level Optimization Pattern Distribution vs. Subject Domain.

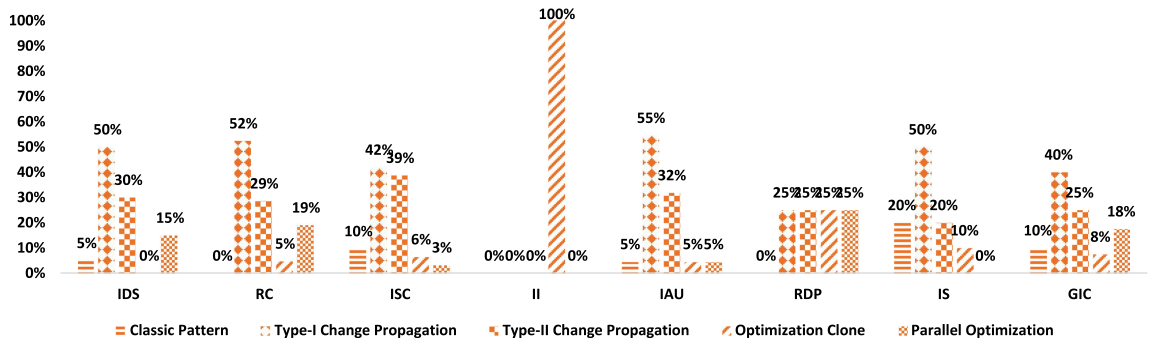


Fig. 10. Design-Level Optimization Pattern Distribution vs. Root Cause.

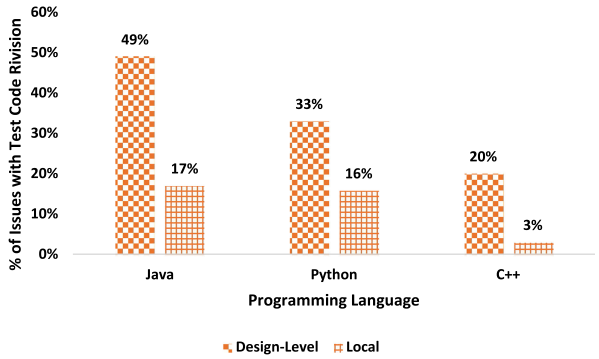


Fig. 11. Test Code Revision in Performance Optimization.

Fig. 11 shows the percentage of design-level and localized optimization that involve test code revision respectively. We separate the analysis of three programming languages. In *Java* issues, 49% of design-level optimization involves test code revision; while only 17% localized optimization involves test code revision. This indicates that design-level optimization is more likely to involve test code change than localized optimization. This observation holds for *Python* and C++ performance issues.

> *RQ-3.1 Implication:* For most (85%) performance issues, the optimization resolution does not include revision of the test cases. When developers apply design-level optimization, they are more likely to revise the test code due to the higher complexity of change. This suggests that it remains an open question whether and how practitioners formally test performance optimization as an integral part of regression testing.

RQ-3.2 What are the common test-production co-change patterns in performance optimization?

We inspect the test code revision to understand the nature of the test and production co-change in performance optimization. We observed five co-change patterns:

1) Performance Input Revision: Developers change the input parameters (e.g. array size and thread waiting time) to performance test cases, such as for stress testing. This is to directly verify the performance optimization. As an example, issue *AVRO-1090* [61] in Fig. 12 is caused by *Redundant Data Processing (RDP)*. The developers added a method named "*avro_raw_array_clear()*" (row 4 in Fig. 12(a)) to clear the array in one go instead of iteratively. The test case increase the input array size by 25 times (row 2 and 3 in Fig. 12(b)) to test the efficiency of clearing a very large array.

```
1     ...
2     avro_resolved_array_writer_free_elements(
3         aiface->child_resolver, self);
4  +  avro_raw_array_clear(&self->children);
5     return 0;
```

(a) Production Code

```
1     test_simple_array_resolved_reader(
2  -      "simple array resolved writer", 10000, ...)
3  +      "simple array resolved writer", 250000, ...)
```

(b) Test Code

Fig. 12. Co-Change Pattern 2: Performance Input Revision.

```
1  +  public Matrix clone(){
2  +      Matrix clone = new Matrix();
3  +      System.arraycopy(single, 0, clone.single);
4  +      return clone;
5  +  }
6
7  -  public Matrix copy(){
8  -      return (Matrix) clone();
9  -  }
10
11    public Matrix multiply(Matrix m1, Matrix m2){
12        ......
13 -      return retVal.copy();
14 +      return retVal.clone();
15    }
```

(a) Production Code

```
1 testMultiplication(){
2    ...
3 -  Matrix m1 = testMatrix.copy();
4 -  Matrix m2 = testMatrix.copy();
5 +  Matrix m1 = testMatrix.clone();
6 +  Matrix m2 = testMatrix.clone();
7    ...
8 }
```

(b) Test Code

Fig. 13. Co-Change Pattern 1: Method Replacement.

2) Method Replacement: A production method is replaced to improve performance; and the test case also replaces this method. For example, issue *PDFBOX-2126* [58] in Fig. 13 is caused by *Inefficiency under Special Case (ISC)*. When rendering a PDF page, the clipping path may change. The original copying method, *Matrix copy()* (row 7 to 9 in Fig. 13(a)), is time and memory consuming in this case. Thus, developers replaced the original method by a deep clone method, *Matrix clone()* (row 1 to 5 in Fig. 13(a)), that directly copies a matrix based on the system's cached value (row 3 in Fig. 13(a)). Consequently, in the test case, developers also replaced the method *Matrix copy()* with *Matrix clone()* for testing the replication of matrix. This saves 50% rendering time on average.

3) Test Logic Modification: The logic in the production changes for performance optimization, and the test code logic changes accordingly. For example, the performance issue, *AVRO-1455* [60] as shown in Fig. 14, is caused by *General Inefficient Computation (GIC)*. Developers modified the returned values to be immutable in the production code to avoid unnecessary primitive type conversion, which saves execution time. Consequently, the respective test case (Fig. 14(b)) adds the checking conditions to test if the returned values are immutable, instead of the original primitive types.

4) Test Case Addition: The production code creates a new method for performance optimization, and the test code adds a respective new test case for this method. As an example, issue *COLLECTIONS-450* [59] is caused by *Inefficiency under Special Cases (ISC)*. The developers add a new method named *forAllButLastDo()*, which executes complicated iteration when the input is not null; otherwise, it directly returns null when the input is null. This method improves performance in the case of null input by avoiding unnecessary iteration. Consequently, the test code creates a new test case named *forAllButLastDoIterator()* to

```
1   public Object getDefaultValue(Field field) {
2     ...
3     case FLOAT:
4   -       return (T)new Float((Float) value);
5   +       return value; // immutable
6     case INT:
7   -       return (T)new Integer((Integer) value);
8   +       return value; // immutable
9     case LONG:
10  -       return (T)new Long((Long) value);
11  +       return value; // immutable
12    case DOUBLE:
13  -       return (T)new Double((Double) value);
14  +       return value; // immutable
15    case BOOLEAN:
16  -       return (T)new Boolean((Boolean) value);
17  +       return value; // immutable
18    ...
19  }
```

(a) Production Code

```
1   public void testDeepCopy() {
2     ...
3     if((field.schema().getType() != Type.ENUM)
4         && field.schema().getType() != Type.NULL)
5   +     && field.schema().getType() != Type.BOOLEAN)
6   +     && field.schema().getType() != Type.INT)
7   +     && field.schema().getType() != Type.LONG)
8   +     && field.schema().getType() != Type.FLOAT)
9   +     && field.schema().getType() != Type.DOUBLE)
10        && field.schema().getType() != Type.STRING)
11    ...
12  }
```

(b) Test Code

Fig. 14. Co-Change Pattern 3: Test Logic Modification.

verify the correctness of *forAllButLastDo()*—whether it can return null when the input is null.

5) Test File Addition: When the production code contains drastic changes with new functions, a new file is added to the test code that is dedicated to testing the new functions. For example, issue *AVRO-709* is caused by *Repeated Computation (RC)*. Developers found that the majority of the execution time was spent on repeatedly looking up *JSON* properties. Thus, developers created two caches to store the searched properties. A set of utility functions is also added to facilitate the implementation of these two caches. Therefore, the developers created a new test file. It not only verifies the correctness of the added functions—such as *read()* and *write()* for *JSON* properties—but also records the execution time of the new functions.

Fig. 15 shows the distribution of the five test-production co-change patterns in issues of the three programming languages. As we can see, *Method Replacement* and *Test Case Addition* are most common in all three languages.

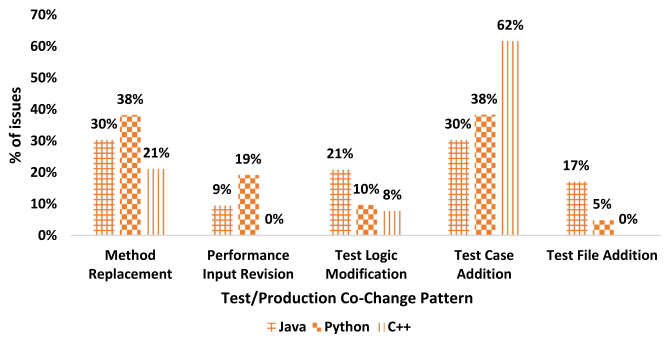


Fig. 15. Distribution of Test/Production Code Co-Change Patterns.

RQ-3.2 Implication: We discover five test/production code co-change patterns in these issues. Among these five patterns, only *Performance Input Revision* directly targets at verifying the performance of the program in different input. The other four patterns indirectly verifying the effectiveness of the performance optimization by focusing on the functional logic of the revised production code. This implies that performance is a potential weak point in the practice of regression testing — developers may lack systematic approach to thoroughly and explicitly test the performance optimization.

5.4 Return on Investment

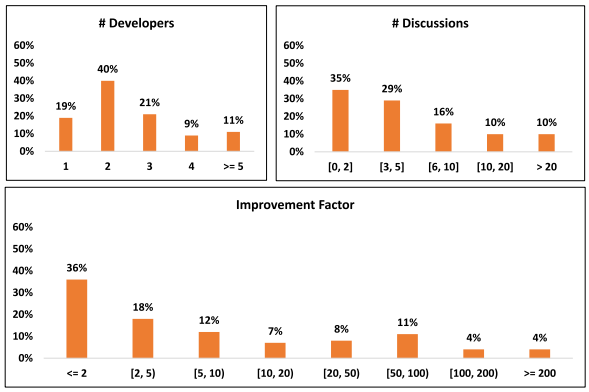
RQ-4.1 What is the overall ROI for addressing performance issues? In particular, what are the other aspects of concerns while fixing performance issues? Fig. 16 shows the results for the Return on Investment (ROI) seprately for the three languages—Java in Fig. 16(a), Python in Fig. 16(b), and C/C++ in Fig. 16(c). The investment is measured by the number of involved developers and the number of discussions. The return is measured by the the extent of performance improvement.

We observe that, regardless of the programming language, the majority performance issues involves less than 2 developers and have less than 5 comments. While the return can vary in a large range up to more than 200 times. As shown in Table 8, the results of *Mann Whitney U Test* suggests that the programming language does not have statistically significant impact on the investment — the *p-values* of the test are all above 0.1.

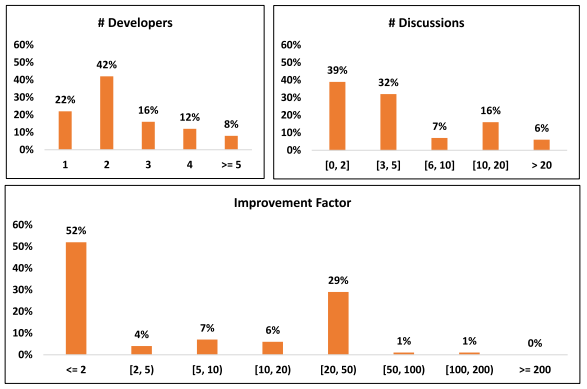
It is worth noting that, as shown in Fig. 16(b), 52% of Python issues yield to less than 2x performance improvements, but 29% of *Python* issues have 20 to 50 times performance improvement. Furthermore, we found that 75% of these issues (with 20 to 50 times improvement) are caused by *Inefficient API Usage (IAU)*. Thus, replacing API in *Python* can result rewarding performance improvement. In comparison, 35% *C++* performance issues yield to more than 50 times performance improvement, and 12% issues even reach more than 200 improvement after fixing.

We found that a total of 46 performance issues (8%) also involve other concerns in seven aspects—maintainability, compatibility, readability, security, flexibility, simplification, and reliability. Fig. 17 shows the distribution of these 46 issues that are associated with the seven concerns for the three languages. As we can see, the most common concern that are often associated with performance optimization is maintainability in all three languages. Overall, the *Mann Whitney U Test* shows that the programming language does not make a difference on the impact to other aspects of concerns. This finding is consistent with our finding in Section 5.2.

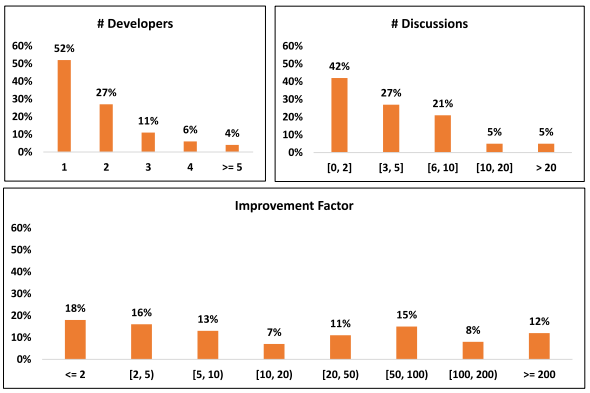
Of a particular note, the influence on the other aspects of concern is not always positive in performance optimization. That is, the resolutions of 85% of the 46 issues yield positive influence on these other concerns; while for the remaining 15% issues, developers trade off other concerns for performance optimization. For example, in issue



(a) *Java* Performance Issues



(b) *Python* Performance Issues



(c) C++ Performance Issues

Fig. 16. Return on Investment (ROI) for Performance Issues.

AVRO-739, developers replaced *JSON* format date time with milliseconds timestamp in *Java* "long" type. The developers acknowledged that this replacement sacrifices readability for trading performance optimization. We only observed negative impacts from performance optimization on maintainability, readability, and simplification in our dataset.

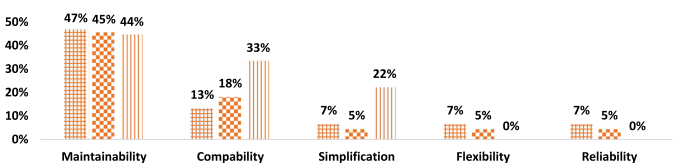


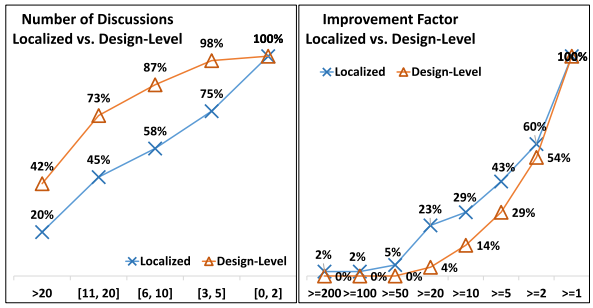
Fig. 17. Other Aspects of Concerns in Performance Optimization.

TABLE 8
Impact of Programming Language on ROI (Measured by *P-Values*)

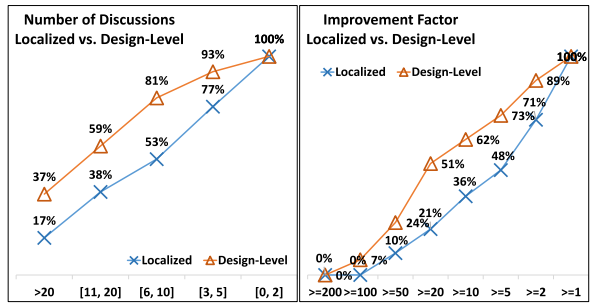
Comparison Group	*Java* vs. *Python*	*Java* vs. C++	*Python* vs. C++
# Involved Developers	0.99	0.75	0.67
# Involved Discussions	0.91	0.83	0.83
Improvement Factor	0.24	0.4	0.11

RQ-4.1 Implication: Regardless of the programming language, it takes less than 2 developers and less than 5 discussions to resolve a performance issue in most cases. We did not observe statistically significant impact from programming language on the return. However, it is worth noting that C++ has the highest potential performance improvement — 12% issues yield to more than 200 times improvement. Furthermore, in 8% performance issues, developers also concern about seven other aspects of quality attributes when fixing the performance issues — maintainability is the most common concern in addressing performance issues.

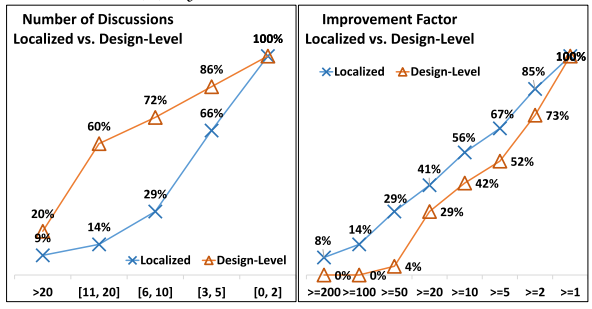
RQ-4.2 How are the ROIs of the localized and design-level optimizations compare to each other? Fig. 18(a), Fig. 18(b), and



(a) *Java* Performance Issues



(b) *Python* Performance Issues



(c) C++ Performance Issues

Fig. 18. ROI for Localized vs. Design-Level Optimization.

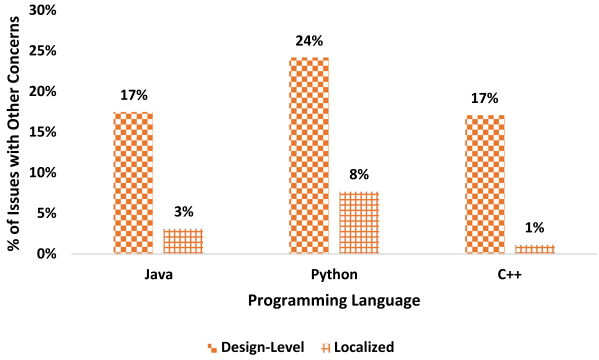


Fig. 19. Other Concerns in Localized vs. Design-Level Optimization .

Fig. 18(c) show the cumulative distribution function plot of the number of discussions (left side) and the improvement factor (right side) for the localized (the lines with the cross marker) and design-level (the lines with the triangle marker) optimization for the issues in *Java*, *Python*, and *C++* subjects. For all three programming languages, the plots of the number of discussions for localized optimizations are consistently and non-trivially below the plots for design-level optimizations. This means that a larger portion of localized optimizations requires less investment compared to the design-level optimizations.

The difference between the improvement factor of localized and design-level optimization is relatively small, compared to the difference in investment. In addition, the result for the three programming languages shows some inconsistency. For *Java* and *C++* issues, the plots of improvement factor for localized optimizations are higher than the plots of design-level optimization—indicating that the localized optimizations tend to yield higher return. But for *Python* issues, the result is the opposite. In depth investigation of the Python data revealed that the majority (60%) of the highly rewarded performance optimization (i.e, improvement factor greater than 20x) involved replacing the API usage at the design-level—replacing cross-cutting API usage simultaneously in multiple Python modules.

Furthermore, as shown in Fig. 19, on average, around 20% of design-level optimizations have other aspects of concerns, but only around 4% localized issues are associated with other aspects of concerns in their return for the three programming languages. Therefore, we infer that one motivating scenario for design-level optimization is for achieving the long-term benefits while fixing performance issues.

> *RQ-4.2 Implication:* Design-level optimization requires more investment to develop compared to localized optimization. But it does not always warrant higher performance improvement. However, for all three programming languages, design-level optimization is more likely to be associated with return in other aspects of concerns, such as maintainability.

6 Limitations and Threats to Validity

First of all, in the data collection, we cannot guarantee that all performance issues in the selected projects are captured. We use keyword matching for selecting performance issues. It is possible that a performance issue report does not contain any keyword. Therefore, we cannot guarantee a 100% recall in retrieving all related issues in the projects. Also, we manually verify an issue truly relates to performance, which relies on our understanding and may involve biases.

In RQ1, the root cause analysis is also potentially biased by the authors' understanding and experience. On the one hand, we cannot guarantee that our dataset comprehensively covers all possible types of performance issues. On the other hand, each root cause may be further categorized into different sub-types. For example, Alam *et al.* categorized *Inefficient Synchronization* performance issues into five detailed situations, such as load imbalance (i.e, disproportionate thread computation and waiting time) and over-synchronization (i.e., extensive lock acquisitions), etc [92]. The goal of this study is to provide an overview of different types of performance issues that are commonly observed in real-life projects. We acknowledge that it is still open to future research to discover a more comprehensive and fine-grained taxonomy of performance issues.

We used the Mann Whitney Test in RQ-1.1, RQ-2.2, and RQ-4.1 for evaluating the significance of the impact of programming language on different aspects of performance issue root causes and resolution. According to [75], p-value < 0.05 (5%) or even p-value < 0.01 (1%) indicates statistically significant difference. No significant impact was found in any of the above analysis — indicating that the root causes, design choices, and ROI are overall independent from the programming language.

In RQ-1.2, we focus on literature that contributes available tools for detecting and fixing real-life performance issues. There are literature that provides a conceptual approach for addressing performance issues without contributing a tool. We did not consider such literature since it is impossible to evaluate the availability and applicability. In addition, we did not evaluate the actual effectiveness and usability of the 42 tools from the literature. The reason is two-fold. First, most of the tools do not have an available link to download. For those with a link, many are no longer maintained. Second, to apply the tools on our dataset requires building and configuring the projects to repeat the performance issues. However, it is a known research challenge to replicate real-life performance issues [1], [4].

In RQ2, the identification of design-level optimization and the related patterns is potentially biased by our understanding, since we manually exclude source file revisions irrelevant to performance optimization, and mark the optimization core in design-level optimization. Furthermore, according to Smith and Williams [15], most performance issues have their roots in poor architectural decisions made before coding is done. Their work is based on a toy project. Our study, to the contrary, is based on 570 performance issues from 13 open source projects. A possible explanation of only 29% design-level optimization in our dataset is that we focus on open source projects. According to RQ-4.2, design-level optimization requires more development effort, and is more likely to involve other aspects of concerns. The choice of design-level optimization could be impacted by various factors, such as the dedication and discipline of developers, project schedule and budget, as well as the need to support other quality attributes such as

maintainability. The 29% design-level optimization — although not significant portions — already underscore the necessity and importance of design-level optimization in practice. In our future work, We plan to explore more performance issues from commercial projects.

In RQ3, a low percentage (15%) of issues are with test code revisions. We conjecture that either the developers mainly focus on revising the production code, neglecting the verification of the performance optimization; or the developers change the related test code in separate commits. If the latter case, it is impractical for us to accurately trace all the related test code changes. Thus, we cannot guarantee that we have retrieved all the related test code revisions. However, this low percentage points to the potential lack of discipline, awareness, and systematic methodology for testing performance issues and their resolution in practice. And we argue that our analysis, based on 85 (15%) issues, should meaningfully represent the production and test code co-change patterns in performance optimization.

Admittedly, in RQ4, the number of involved developers and the number of discussions may not accurately measure the effort. In proprietary projects, the effort could be measured more accurately based on the reported human hours [106]. However, humans hours are not available in our research, since it is based on open source software projects. In addition, there are other measures, such as LoC (Lines of Code). However, according to Anda *et al.* [107], LoC could be misleading in representing the human effort in software development. For example, a large amount of LoC does not imply more effort. The two metrics employed in our study may provide a closer proxy compared to the LoC, and they have been commonly used in previous studies, such as [108], [109].

In addition, the performance improvement is based on the profiling data provided in the performance issue reports and their discussions. We did not reproduce the issues to collect the profiling data ourselves. It is a known challenge to replicate real-life performance issues due to various factors, such as reproducing the run-time environment, and configuring the input that lead to the performance issues [110]. Reproducing the performance issues in large-scale, distributed software systems, such as *Beam* and *Mesos*, is particularly impractical in lab settings. We lack a systematic approach to collect reliable profiling data. Thus, we decide to use the reported performance improvement, which is more reliable due to developers' expertise with their projects. Reproducing the performance issues is highly valuable and also a rich problem that deserves to be investigated in a separate, dedicated study.

Finally, the availability of discussions that are related to other aspects of concerns (i.e. the form of return other than performance improvement) relies on the expertise of and convention following by the developers. That is, we cannot verify the benefits to other aspects of concerns mentioned by the developers; and, we cannot guarantee that developers always acknowledge such benefits in all the issue discussions when applicable.

7 Related Work

7.1 Software Performance Empirical Study

There are several prior empirical studies that also investigate and categorize real-life software performance issues [6], [7], [8], [9]. However, these studies are not as comprehensive as ours in terms of the size and diversity of the dataset. In addition, none of the prior studies investigates the design-level optimization, the test-production co-changes, or the ROI in resoving performance issues.

Jin *et al.* reviewed 109 real-life performance issues from 5 widely-used open source software projects (e.g., *Mozilla*, *MySQL*, and *Chromium*, etc.). They categorized the root cause of these performance issues into four types, including: 1) "*Uncoordinated Functions*," which map to the *Repeated Computation* in our study, 2) "*Skippable Functions*," which map to the *General Inefficient Computation* in our work, 3) "*Synchronization Issues*," which map to our *Inefficient Synchronization*, and 4) "*Others*," which are the issues that cannot be categorized into the above three.

Liu *et al.* [8] and Linares-Vasquez *et al.* [7] focused on performance issues from *Android* smart-phone applications. Liu *et al.* [8] categorized performance issues into three types based on their consequences, including 1) GUI lagging, 2) energy leak, and 3) memory bloat. Linares-Vasquez *et al.* [7] summarized three root causes for performance issues that related to energy consumption, including: 1) "*API misuse*," 2) "*data structure bad manipulation*," and 3) "*failure of switching out of thread*". These root causes are equivalent to the *Inefficient API Usage*, *Inefficient Data Structure*, and *Inefficient Synchronization* in our study.

Selakovic *et al.* [9] investigated 98 fixed performance issues from 16 *JavaScript*-based projects and summarized seven types of performance issue root causes. Comparing to our findings, they only miss the *Inefficient Synchronization* since *JavaScript* does not support multi-threading.

7.2 Model-Based Performance Engineering

Model-based performance Engineering uses modelling techniques to predict performance metrics of a system, such as execution speed, resource utilization, and throughput [111], [112], [113]. In ongoing software projects, practitioners often recover architecture models from the source code of a software system using reverse engineering techniques [114], [115], [116]. There are three types of architecture models that are often used for model-based performance engineering. They include 1) the behavioral model of the system, which captures architecture as the collaborations among system objects, their internal state changes, the dynamic interactions among objects during system operation, e.g., *UML Sequence Diagram* [117], [118], 2) the component model, which captures the architecture of a system as a set of components, connectors, and their compositions, e.g., *Palladio Component Model* [119], [120], and 3) the hybrid model, which captures the architecture of a system as the combination of the component structures and their behavioral interactions, e.g., *Probability Matrix* [121], [122]. Practitioners annotate these architecture models with performance-related information, such as branching probabilities [123], [124], workload status [121], [122], [125], and resource demand [114], [116], [120]. The annotated architecture models are transformed into performance models, such as *Queuing Networks* [117], [118], [126], [127], [128], *Place/Transition (Petri) Nets* [111], [129], [130], [131], and *Stochastic Process Algebra* [2], [111], [132]. Finally, these performance models help to predict the performance metrics of a system by using the analytical tools, such as *SHARPE* [123], [124], [133], *QPN Solver* [129], and *GreatSPN* [2], [134].

Model-based Performance Engineering can predict the performance of a system early—even before the implementation effort starts [2]. However, it requires high expertise from the analyst, and thus is potentially difficult to scale with the complexity of real-life systems.

7.3 Performance Testing and Profiling

Performance testing is one of the most thoroughly studied approaches to addressing performance concerns in practice. Performance testing executes a system and constructs a profile of the system, in terms of responsiveness and stability under various workloads [1].

There are four major types of performance testing methodologies, including load testing, stress testing, endurance testing, and spike testing [17], [18]. Load testing evaluates the behavior of a software system under specific workloads. Stress testing executes and profiles the system under extreme workloads to discover the maximum capacity of the system. Endurance testing executes and profiles the system under continuous workload. The purpose is to determine whether the system can scale up to support enduring and increasing workloads. Spike testing determines whether a system can sustain a sudden increase in workload. [19]. During performance testing, practitioners leverage profiling tools to keep track of performance metrics, such as response time, throughput, and resource utilization [135]. These tools are available for different platforms, programming languages, and execution environments, with different advantages [136]. For example, *WebLoad* can generate real-life and reliable workload scenarios for testing complex systems [137]. *LoadNinja* has the highest coverage for performance testing [138]. *LoadView* can be applied to real-life browsers and web applications [139]. *StresStimulus* can detect hidden concurrency errors by measuring performance metrics, such as network latency and data transmission loss ratio [140]. *Apache JMeter* [141] is the most widely-used performance testing/profiling tool for *Java* projects and it has been integrated to many IDEs, such as *Eclipse* and *NetBeans*. *SmartMeter* can automatically generate a performance assessment report [142]. *Rational Performance Tester* [143] is a powerful performance testing tool developed by *IBM*, which supports load testing that involves multiple users and generates a comprehensive performance assessment report.

One purpose of performance testing is to reveal performance bottlenecks in software systems by dynamically testing the system [17], [144], [145]. However, it does not provide insights regarding how real-life performance issues are usually caused and should be resolved in practice. Therefore, this empirical study provides complementary knowledge for practitioners to more effectively treat performance issues.

8 Conclusion

This study contributes a large-scale empirical study of 570 real-life performance issues from Java, C/C++, and Python projects. This study is the first-of-its-kind to provide holistic analysis of the root causes and resolutions of these performance issues in that it covers the technical, engineering, and economic perspectives in resolving performance issues.

First, we revealed **eight** common root causes and corresponding resolutions of performance issues. These eight root causes apply generally to three programming languages — *Java*, *Python*, and *C++*. The programming language does not have statistically significant impact on the root causes. Practitioners should be aware of these common root causes and resolutions in preventing and resolving performance issues.

Second, to the best of our knowledge, we are the first to investigate performance optimization from a design-perspective. That is, we found that some performance issues require coordinated revision of a group of related source files and their design structure to achieve performance improvement and other aspects of quality concerns, such as maintainability. We revealed four common design-level optimization strategies that practitioners should be aware of.

Next, we consider testing as an integral part of performance optimization by analyzing the common test-production co-change patterns in performance optimization. We found that developers not only directly verify the performance improvement by tuning input sizes, but more frequently indirectly verify the performance optimization by focusing on the functional logic. We infer that practitioners may lack formal processes to follow in verifying performance optimization in regression testing.

Finally, we consider software development as an economic activity by evaluating the ROI of performance optimization. In particular, we found that the design-level optimization usually requires higher investment — more developers and more discussions to come up with the resolution. But it does not always yield to higher extent of performance improvement. However, design-level optimization is necessary when practitioners also concern about other aspects of quality concerns, such as maintainability.

Therefore, this study provides unique and novel insights for practitioners to treat performance issues from the technical, engineering, and economic perspectives, which are not available in the state-of-the-art research.

References

[1] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, vol. 1. Reading, MA, USA: Addison-Wesley, 2002.

[2] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, " Model-based performance prediction in software development: A survey," *IEEE Trans. Softw. Eng.*, vol. 30, no. 5, pp. 295–310, May 2004.

[3] G. (H.) Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 151–160.

[4] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *Proc. 9th IEEE Work. Conf. Mining Softw. Repositories*, 2012, pp. 199–208.

[5] G. Xu, D. Yan, and A. Rountev, "Static detection of loop-invariant data structures," in *Proc. Eur. Conf. Object-Oriented Program.*, 2012, pp. 738–763.

[6] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN Notices*, vol. 47 no. 6, pp. 77–88, 2012.

[7] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy API usage patterns in Android apps: An empirical study," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 2–11.

[8] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1013–1024.

[9] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: An empirical study," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 61–72.

[10] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 902–912.

[11] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *Proc. 37th Int. Conf. Softw. Eng.*, 2013, pp. 562–571.

[12] M. Dhok and M. K. Ramanathan, "Directed test generation to detect loop inefficiencies," in *Proc. 24th Int. Symp. Found. Softw. Eng.*, 2016, pp. 895–907.

[13] L. Song and S. Lu, "Performance diagnosis for inefficient loops," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 370–380.

[14] O. Olivo, I. Dillig, and C. Lin, "Static detection of asymptotic performance bugs in collection traversals," in *Proc. ACM SIGPLAN Notices*, 2015, pp. 369–378.

[15] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Proc. 2nd Int. Workshop Softw. Perform.*, 2000, pp. 127–136.

[16] [Online]. Available: https://issues.apache.org/jira/browse/avro-753

[17] L. Bass, P. Clements, and R. Kazman, *Softw. architecture in Pract.* Addison-Wesley Professional, 2003.

[18] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, pp. 237–246, 2013.

[19] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Proc. Future of Softw. Eng.*, 2007, pp. 171–187,.

[20] Y. Zhao, Lu Xiao, X. Wang, L. Sun, B. Chen, Y. Liu, and A. B. Bondi, "How are performance issues caused and resolved?-An empirical study from a design perspective," in *Proc. 17th Int. Conf. Perform. Eng.*, 2020, pp. 181–192.

[21] K. Kumar and S. Dahiya, "Programming languages: A survey," *Int. J. Recent Innov. Trends Comput. Commun.*, vol. 5 no. 5, pp. 307–313, 2017.

[22] [Online]. Available: https://www.tiobe.com/tiobe-index/

[23] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*, vol. 1. Cambridge, MA, USA: MIT Press, 2000.

[24] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: A new form of architecture insight," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 967–977.

[25] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," in *Proc. ACM SIGSOFT Softw. Eng. Notes*, 2001, pp. 99–108.

[26] X. Wang, L. Xiao, K. Huang, B. Chen, Y. Zhao, and Y. Liu, "DesignDiff: Continuously modeling software design difference from code revisions," in *Proc. IEEE Int. Conf. Softw. Archit.*, 2020, pp. 179–190.

[27] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *Proc. 6th ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 395–404.

[28] R. R. Sambasivan *et al.*, "Diagnosing performance changes by comparing request flows," in *Proc. 8th USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 1–14.

[29] B. Chen, Y. Liu, and W. Le, "Generating performance distributions via probabilistic symbolic execution," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 49–60.

[30] [Online]. Available: https://github.com/apache/pdfbox

[31] [Online]. Available: https://github.com/apache/avro

[32] [Online]. Available: https://github.com/apache/groovy

[33] [Online]. Available: https://github.com/apache/commons-collections

[34] [Online]. Available: https://github.com/apache/ant-ivy

[35] [Online]. Available: https://github.com/apache/beam

[36] [Online]. Available: https://github.com/apache/qpid-python

[37] [Online]. Available: https://github.com/apache/libcloud

[38] [Online]. Available: https://github.com/apache/climate

[39] [Online]. Available: https://github.com/svn2github/pylucene

[40] [Online]. Available: https://github.com/php/php-src

[41] [Online]. Available: https://github.com/apache/kudu

[42] [Online]. Available: https://github.com/apache/mesos

[43] [Online]. Available: https://projects.apache.org/projects.html?category

[44] M. Pradel, M. Huggler, and T. R. Gross, "Performance regression testing of concurrent classes," in *Proc. 23th Int. Symp. Softw. Testing Anal.*, 2014, pp. 13–25.

[45] Z. Chen *et al.*, "Speedoo: Prioritizing performance optimization opportunities," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 811–821.

[46] [Online]. Available: https://issues.apache.org/jira/browse/pdfbox-591

[47] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 120–131.

[48] [Online]. Available: https://issues.apache.org/jira/browse/pdfbox-1337

[49] B. Kitchenham, "Procedures for performing systematic reviews," Keele Univ., Keele, UK, 2004.

[50] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proc. 18th Int. Conf. Eval. Assessment Softw. Eng.*, 2014, pp. 1–10.

[51] S. Jalali and C. Wohlin, "Systematic literature studies: Database searches versus. backward snowballing," in *Proc. Int. Symp. Empir. Softw. Eng. Meas.*, 2012, pp. 29–38.

[52] D. Costa and A. Andrzejak, "Collectionswitch: A framework for efficient and dynamic collection selection," in *Proc. Int. Symp. Code Gener. Optim.*, 2018, pp. 16–26.

[53] O. Shacham, M. Vechev, and E. Yahav, "Chameleon: Adaptive selection of collections," in *Proc. ACM SIGPLAN Notices*, 2009, pp. 408–418.

[54] G. Xu, "COCO: Sound and adaptive replacement of java collections," in *Proc. 27th Eur. Conf. Object-Oriented Program.*, 2013, pp. 1–26.

[55] G. Xu and A. Rountev, "Detecting inefficiently-used containers to avoid bloat," in *Proc. 31st Int. Conf. Program. Lang. Des. Implementation*, 2010, pp. 160–173.

[56] Y. Zhao, Lu Xiao, X. Wang, Z. Chen, B. Chen, and Y. Liu, "Butterfly space: An architectural approach for investigating performance issues," in *Proc. 17th Int. Conf. Softw. Archit.*, 2020, pp. 202–213.

[57] [Online]. Available: https://issues.apache.org/jira/browse/pdfbox-2303

[58] [Online]. Available: https://issues.apache.org/jira/browse/pdfbox-2126

[59] [Online]. Available: https://issues.apache.org/jira/browse/collections-450

[60] [Online]. Available: https://issues.apache.org/jira/browse/avro-1455

[61] [Online]. Available: https://issues.apache.org/jira/browse/avro-1090

[62] [Online]. Available: https://issues.apache.org/jira/browse/pdfbox-410

[63] [Online]. Available: https://issues.apache.org/jira/browse/libcloud-254

[64] [Online]. Available: https://issues.apache.org/jira/browse/mesos-2126

[65] L. D. Toffola, M. Pradel, and T. R. Gross, "Performance problems you can fix: A dynamic analysis of memoization opportunities," in *Proc. ACM SIGPLAN Notices*, 2015, pp. 607–622.

[66] D. Kawrykow and M. P. Robillard, "Detecting inefficient API usage," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 183–186.

[67] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "API method recommendation without worrying about the task-API knowledge gap," in *Proc. 33rd Int. Conf. Automated Softw. Eng.*, 2018, pp. 293–304.

[68] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proc. ACM SIGSOFT Softw. Eng. Notes*, 2005, pp. 306–315.

[69] D. Kawrykow and M. P. Robillard, "Improving API usage through automatic detection of redundant code," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 111–122.

[70] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating performance bottleneck detection using search-based application profiling," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 270–281.

[71] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically generating pathological inputs," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 254–265.

[72] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," *ACM SIGPLAN Notices*, vol. 47 no. 6, pp. 89–98, 2012.

[73] M. Brünink and D. S. Rosenblum, "Mining performance specifications," in *Proc. 24th Int. Symp. Found. Softw. Eng.*, 2016, pp. 39–49.

[74] [Online]. Available: https://issues.apache.org/jira/browse/pdfbox-600

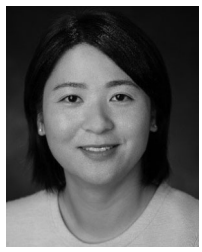
[75] P. E. McKnight and J. Najab, "Mann-Whitney U test," *The Corsini Encyclopedia of Psychology*, Hoboken, NJ, USA: Wiley, 2010, p. 1.

[76] [Online]. Available: https://issues.apache.org/jira/browse/pdfbox-604

[77] L. Liu and S. Rus, "Perflint: A context sensitive performance advisor for C++ programs," in *Proc. Int. Symp. Code Gener. Optim.*, 2009, pp. 265–274.
[78] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande, "Brainy: Effective selection of data structures," in *Proc. ACM SIGPLAN Notices*, 2011, pp. 86–97.
[79] J. Gil and Y. Shimron, "Smaller footprint for java collections," in *Proc. Eur. Conf. Object-Oriented Program.*, 2012, pp. 356–382.
[80] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proc. 39th Int. Conf. Parallel Process. Workshops*, 2010, pp. 207–216.
[81] K. Nguyen and G. Xu, "CACHETOR: Detecting cacheable data to remove bloat," in *Proc. Proc. 9th ACM SIGSOFT Symp. Found. Softw. Eng.*, 2013, pp. 268–278.
[82] S. Wen, M. Chabbi, and X. Liu, "REDSPY: Exploring value locality in software," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017, pp. 47–61.
[83] P. Su, S. Wen, H. Yang, M. Chabbi, and X. Liu, "Redundant loads: A software inefficiency indicator," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 982–993.
[84] E. Albert, M. Gómez-Zamalloa, and G. Puebla, "PET: A partial evaluation-based test case generation tool for Java bytecode," in *Proc. ACM SIGPLAN Workshop Partial Eval. Prog. Manipulation*, 2010, pp. 25–28.
[85] J. Diemer, P. Axer, and R. Ernst, "Compositional performance analysis in Python with pyCPA," *Proc. WATERS*, 2012, Art. no. 46.
[86] G. L. Lee, D. H. Ahn, B. R. de Supinski, J. Gyllenhaal, and P. Miller, "Pynamic: The Python dynamic benchmark," in *Proc. 10th Int. Symp. Workload Characterization*, 2007, pp. 101–106.
[87] I. Manotas, L. Pollock, and J. Clause, "SEEDS: A software engineer's energy-optimization decision support framework," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 503–514.
[88] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic, "LIME: A framework for debugging load imbalance in multi-threaded execution," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 201–210.
[89] T. Liu and E. D. Berger, "SHERIFF: Precise detection and automatic mitigation of false sharing," *ACM SIGPLAN Notices*, vol. 46 no. 10, pp. 3–18, 2011.
[90] T. Liu, C. Tian, Z. Hu, and E. D. Berger, "PREDATOR: Predictive false sharing detection," *ACM SIGPLAN Notices*, vol. 49, pp. 3–14, 2014.
[91] T. Yu and M. Pradel, "SyncProf: Detecting, localizing, and optimizing synchronization bottlenecks," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 389–400.
[92] M. Mejbah Ul Alam, T. Liu, G. Zeng, and A. Muzahid, "SyncPerf: Categorizing, detecting, and diagnosing synchronization performance bugs," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 298–313.
[93] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in Python," *IEEE Access*, vol. 8, pp. 89497–89509, 2020.
[94] A. Gocht, R. Schöne, and J. Frenzel, "Advanced Python performance monitoring with score-p," in *Tools for High Performance Computing 2018/2019*. Berlin, Germany: Springer, 2021, pp. 261–270.
[95] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Proc. ACM/IEEE Conf. SuperComput.*, 2003, pp. 21–21.
[96] R. T. J. Ramos, A. R. Carneiro, V. Azevedo, M. P. Schneider, D. Barh, and A. Silva, "Simplifier: A web tool to eliminate redundant NGS Contigs," *Bioinformation*, vol. 8, no. 20, 2012, Art. no. 996.
[97] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Trans. Softw. Eng.*, vol. 42 no. 12, pp. 1148–1161, Dec. 2016.
[98] Qi Luo, D. Poshyvanyk, A. Nair, and M. Grechanik, "Forepost: A tool for detecting performance problems with feedback-driven learning software testing," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 593–596.
[99] O. Javed *et al.*, "PerfCI: A toolchain for automated performance testing during continuous integration of Python projects," in *Proc. 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 1344–1348.
[100] A. S. Siqueira, R. C. da Silva, and L.-R. Santos, "Perprof-py: A python package for performance profile of mathematical optimization software," *J. Open Res. Softw.*, vol. 4, no. 1, 2016, Art. no. e12.
[101] J. S. Stein, W. F. Holmgren, J. Forbess, and C. W. Hansen, "PVLIB: Open source photovoltaic performance modeling functions for Matlab and Python," in *Proc. 43rd Photovoltaic Specialists Conf.*, 2016, pp. 3425–3430.
[102] J. H. Dawes, M. Han, O. Javed, G. Reger, G. Franzoni, and A. Pfeiffer, "Analysing the performance of Python-based web services with the vypr framework," in *Proc. 20th Int. Conf. Runtime Verification*, 2020, pp. 67–86.
[103] [Online]. Available: https://issues.apache.org/jira/browse/pdfbox-893
[104] [Online]. Available: https://issues.apache.org/jira/browse/pdfbox-3421
[105] [Online]. Available: https://issues.apache.org/jira/browse/pdfbox-3224
[106] J. Xiao, Q. Wang, M. Li, Ye Yang, F. Zhang, and L. Xie, "A constraint-driven human resource scheduling method in software development and maintenance process," in *Proc. 36th Int. Conf. Softw. Maintenance*, 2008, pp. 17–26.
[107] B. Anda, H. Dreiem, D. IK Sjøberg, and M. Jørgensen, "Estimating software development effort based on use cases—Experiences from industry," in *Proc. 2nd Int. Conf. Unified Model. Lang.*, 2001, pp. 487–502.
[108] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. Van Deursen, "Communication in open source software development mailing lists," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 277–286.
[109] M. Staron and W. Meding, "Monitoring bottlenecks in agile and lean software development projects–a method and its industrial use," in *Proc. 12th Int. Conf. Product Focused Softw. Process Improvement*, 2011, pp. 3–16.
[110] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk, "Generating reproducible and replayable bug reports from android application crashes," in *Proc. 23rd Int. Conf. Prog. Comprehension*, 2015, pp. 48–59.
[111] F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, and S. Kounev, "Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures," *IEEE Trans. Softw. Eng.*, vol. 41, no. 2, pp. 157–175, Feb. 2014.
[112] C. U. Smith and L. G. Williams, "Software performance engineering," in *UML for Real*. Berlin, Germany: Springer, 2003, pp. 343–365.
[113] C. U. Smith, "Introduction to software performance engineering: Origins and outstanding problems," in *Proc. Int. Sch. Formal Methods Des. Comput., Commun. Softw. Syst.*, 2007, pp. 395–428.
[114] F. Brosig, N. Huber, and S. Kounev, "Modeling parameter and context dependencies in online architecture-level performance models," in *Proc. 15th ACM SIGSOFT Symp. Compon. Based Softw. Eng.*, 2012, pp. 3–12.
[115] C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models," in *Proc. 2nd ACM/SPEC Int. Conf. Perform. Eng.*, 2011, pp. 19–30.
[116] H. Groenda, "Improving performance predictions by accounting for the accuracy of composed performance models," in *Proc. 8th Int. ACM SIGSOFT Conf. Qual. Softw. Archit.*, 2012, pp. 111–116.
[117] D. Arcelli, V. Cortellessa, A. Filieri, and A. Leva, "Control theory for model-based performance-driven software adaptation," in *Proc. 11th Int. ACM SIGSOFT Conf. Qual. Softw. Archit.*, 2015, pp. 11–20.
[118] D. Arcelli and V. Cortellessa, "Software model refactoring based on performance analysis: Better working on software or performance side?," in *Proc. 10th Formal Eng. Approaches Softw. Compon. Archit.*, 2013, pp. 33–47.
[119] A. Martens, H. Koziolek, L. Prechelt, and R. Reussner, "From monolithic to component-based performance evaluation of software architectures," *Empir. Softw. Eng.*, vol. 16, no. 5, pp. 587–622, 2011.
[120] Q. Noorshams, R. Reeb, A. Rentschler, S. Kounev, and R. Reussner, "Enriching software architecture models with statistical models for performance prediction in modern storage environments," in *Proc. 17th Int. Symp. Compon.-Based Softw. Eng.*, 2014, pp. 45–54.
[121] Steffen Becker, "Coupled model transformations for QoS enabled component-based software design," PhD dissertation, Dept. Comput. Sci., Univ. Oldenburg, Oldenburg, Germany, 2008.
[122] S. Balsamo and M. Marzolla, "Performance evaluation of UML software architectures with multiclass queueing network models," in *Proc. 5th Int. Workshop Softw. Perform.*, 2005, pp. 37–42.

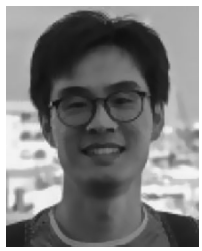
[123] S. S. Jalali, H. Rashidi, and E. Nazemi, "A new approach to evaluate performance of component-based software architecture," in *Proc. 5th Eur. Symp. Comput. Model. Simul.*, 2011, pp. 451–456.
[124] V. S. Sharma and K. S. Trivedi, "Architecture based analysis of performance, reliability and security of software systems," in *Proc. 5th Int. Workshop Softw. Perform.*, 2005, pp. 217–227.
[125] R. Eramo, V. Cortellessa, A. Pierantonio, and M. Tucci, "Performance-driven architectural refactoring through bidirectional model transformations," in *Proc. 8th Int. Conf. Qual. Softw. Archit.*, 2012, pp. 55–60.
[126] N. Mani, D. C. Petriu, and M. Woodside, "Towards studying the performance effects of design patterns for service oriented architecture," in *Proc. 2nd ACM/SPEC Int. Conf. Perform. Eng.*, 2011, pp. 499–504.
[127] A. Di Marco and P. Inverardi, "Compositional generation of software architecture performance QN models," in *Proc. 4th Work. Int. Conf. Softw. Archit.*, 2004, pp. 37–46.
[128] X. Wu, Y. Liu, and I. Gorton, "Exploring performance models of Hadoop applications on cloud architecture," in *Proc. 11th Int. Conf. Qual. Softw. Archit.*, 2015, pp. 93–101.
[129] S. Eismann, J. Grohmann, J. Walter, J. Von Kistowski, and S. Kounev, "Integrating statistical response time models in architectural performance models," in *Proc. 16th Int. Conf. Softw. Archit.*, 2019, pp. 71–80.
[130] J. Walter, C. Stier, H. Koziolek, and S. Kounev, "An expandable extraction framework for architectural performance models," in *Proc. 8th Int. Conf. Perform. Eng.*, 2017, pp. 165–170.
[131] C. U. Smith, C. M. Lladó, V. Cortellessa, A. D. Marco, and L. G. Williams, "From UML models to software performance results: An SPE process based on XML interchange formats," in *Proc. 5th Int. Workshop Softw. Perform.*, 2005, pp. 87–98.
[132] S. Balsamo, M. Marzolla, A. Di Marco, and P. Inverardi, "Experimenting different software architectures performance techniques: A case study," in *Proc. 4th Int. Workshop Softw. Perform.*, 2004, pp. 115–119.
[133] X. Zhang and C.-H. Lung, "Improving software performance and reliability with an architecture-based self-adaptive framework," in *Proc. 34th Annu. Comput. Softw. Appl. Conf.*, 2010, pp. 72–81.
[134] B. Li, Li Liao, and Y. Cheng, "Evaluating software architecture evolution using performance simulation," in *Proc. 4th Int. Conf. Appl. Comput. Informat. Technol.*, 2016, pp. 7–13.
[135] M. C. Patel and R. Gulati, "Software performance testing tools–A comparative analysis," *Int. J. Eng. Res. Dev*, vol. 3, no. 9, pp. 58–61, 2012.
[136] K. M. Mustafa, R. E. Al-Qutaish, and M. I. Muhairat, "Classification of software testing tools based on the software testing methods," in *Proc. 2nd Int. Conf. Comput. Elect. Eng.*, 2009, pp. 229–233.
[137] W. Sobel *et al.*, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," in *Proc. 27th Int. Conf. Consulting Actuaries*, 2008, Art. no. 228.
[138] N. Srivastava, U. Kumar, and P. Singh, "Software and performance testing tools," *J. Inform. Elect. Electron. Eng.*, vol. 2, no. 01, pp. 1–12, 2021.
[139] L. S. Andrés, S. Phillips, and D. Childs, "A water-lubricated hybrid thrust bearing: Measurements and predictions of static load performance," *J. Eng. Gas Turbines Power*, vol. 139, no. 2, 2017, Art. no. 022506.
[140] P. Markey and G. Clynch, "A performance analysis of WS-* (SOAP) and restful web services for implementing service and resource orientated architectures," in *Proc. Conf. 12th Informat. Technol. Telecommun.*, 2013, Art. no. 93.
[141] E. H. Halili, *Apache JMeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for Your Websites*. Birmingham, U.K.: Packt Publishing Ltd., 2008.
[142] S. Prasad and S. B. Avinash, "Smart meter data analytics using openTSDB and Hadoop," in *Proc. IEEE Innov. Smart Grid Technol.-Asia*, 2013, pp. 1–6.
[143] J. Urković, J. Trninić, and V. Vuković, "Test software functionality, but test its performance as well," *Manage. Informat. Syst.*, vol. 6, no. 2, pp. 3–7, 2011.
[144] A. B. Bondi, *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*. London, U.K.: Pearson Education, 2015.
[145] A. B. Bondi, "Best practices for writing and managing performance requirements: A tutorial," in *Proc. 3rd ACM/SPEC Int. Conf. Perform. Eng.*, 2012, pp. 1–8.

Yutong Zhao is currently working toward the PhD degree with the School of Systems and Enterprises, Stevens Institute of Technology. His research focuses on software architecture and performance. He is advised by Dr. Lu Xiao.



Lu Xiao received the PhD degree in computer science from Drexel University in 2016, advised by Dr. Yuanfang Cai. She is an assistant professor with the School of Systems and Enterprises, Stevens Institute of Technology. Her research focuses on software architecture, software evolution, and maintenance. In particular, she is interested in modeling and analyzing software architecture and its evolution for addressing quality problems, such as maintenance quality and performance.

Andre B. Bondi received the BSc in mathematics from the University of Exeter, the MSc degree in statistics from University College London, and the MS and PhD in computer science from Purdue University. He is an adjunct professor with the School of Systems and Enterprises at Stevens Institute of Technology. He was with Siemens Corporate Technologies for 12 years. He has held senior performance positions at two startup companies. He spent thirteen years with AT&T Labs and its predecessor, Bell Labs. Prior to joining Bell Labs, he was an assistant professor of Computer Science with the University of California, Santa Barbara. He has nine U.S. patents.



Bihuan Chen received the bachelor's and PhD degrees in computer science from Fudan University, in 2009 and 2014, respectively. He is an associate professor with Fudan University. Currently he is a postdoctoral research fellow with Nanyang Technological University. His research currently focuses on self-adaptive systems, program analysis, and software testing.

Yang Liu received the bachelor of computing (Honours) from the National University of Singapore (NUS), in 2005and the PhD degree in 2010, and started his postdoctoral work in NUS, MIT, and SUTD. In 2011, he is awarded the Temasek Research Fellowship with NUS to be the principal investigator in the area of Cyber Security. In 2012 fall, he joined Nanyang Technological University (NTU) as a Nanyang assistant professor. He is currently a full professor and the director with the cybersecurity lab, NTU. He specializes in software verification, security and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. His work led to the development of a state-of-the-art model checker, Process Analysis Toolkit (PAT). By now, he has more than 300 publications and six best paper awards in top tier conferences and journals. With more than 20 million Singapore dollar funding support, he is leading a large research team working on the state-of-the-art software engineering and cybersecurity problems.