How Are Performance Issues Caused and Resolved?—An Empirical Study from a Design Perspective

Yutong Zhao[1], Lu Xiao[1], Xiao Wang[1], Lei Sun[1], Bihuan Chen[2], Yang Liu[3], Andre B. Bondi[1]
yzhao102,lxiao6,xwang97@stevens.edu;bhchen@fudan.edu.cn;yangliu@ntu.edu.sg;abondi@stevens.edu
Stevens Institute of Technology [1], Fudan University[2], Nanyang Technological University[3]

ABSTRACT

Empirical experience regarding how real-life performance issues are caused and resolved can provide valuable insights for practitioners to effectively and efficiently prevent, detect, and fix performance issues. Prior work shows that most performance issues have their roots in poor architectural decisions. This paper contributes a large scale empirical study of 192 real-life performance issues, with an emphasis on software design. First, this paper contributes a holistic view of eight common root causes and typical resolutions that recur in different projects, and surveyed existing literature, in particular, tools, that can detect and fix each type of performance issue. Second, this study is first-of-its-kind to investigate performance issues from a design perspective. In the 192 issues, 33% required design-level optimization, i.e. simultaneously revising a group of related source files for resolving the issues. We reveal four design-level optimization patterns, which have shown different prevalence in resolving different root causes. Finally, this study investigated the Return on Investment for addressing performance issues, to help practitioners choose between localized or design-level optimization resolutions, and to prioritize issues due to different root causes.

CCS CONCEPTS

• **Software and its engineering → Software design tradeoffs**; **Software design engineering**.

KEYWORDS

software performance, software design structure, design patterns

1 INTRODUCTION

Performance issues can result in long execution time, memory bloat, and even program crash [1]. Prior work by Smith and Williams [2–5] shows that most performance issues have their roots in poor architectural decisions made before coding is done. This paper presents an empirical study regarding how real-life performance issues are caused and resolved, with an emphasis on the design structure of software systems. The findings in this paper can provide valuable insights for practitioners to effectively and efficiently prevent, detect, and fix performance issues.

This paper investigated 192 real-life performance issues from five popular open source projects implemented in Java. We identified eight types of recurring root causes and their corresponding resolutions. Through extensive literature review, we found that different types of performance issues were investigated separately in previous studies, and there are detecting and/or fixing tools available for Java and C/C++ projects [6–27]. However, only a very limited amount of research has provided a holistic view of different types of performance issues and their prevalence. Selakovic et al.'s work [28] is a most recent example, which revealed seven types of performance issues based on 98 performance issues exclusively from JavaScript projects. The findings in this paper generally agree with Selakovic et al.'s findings. However, we observed that the prevalence of different root causes are impacted by different factors, including, but not limited to, programming languages and project domains.

To the best of our knowledge, this study is the first to investigate the resolutions of performance issues from a design perspective. In existing literature, performance issues are usually treated by localized code revisions, i.e. a few lines of code revisions in a single source file. However, many proprietary projects might have encountered 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 [29]. This study underscores this finding in open source projects—in the 192 issues, 33% are addressed by design-level optimization. We revealed four typical patterns of design-level optimization that are necessary for some performance issues: 1) classic design patterns, where developers employ classic design patterns to improve performance and achieve good design; 2) change propagation, where the core performance optimization propagates changes to a group of structurally related source files; 3) optimization clone, where developers clone the same performance optimization in multiple locations of the code base; and 4) parallel optimization, where developers make independent optimizations in parallel for addressing a performance issue. These patterns are necessary to treat architectural performance anti-patterns. For example, in issue *AVRO-753* (discussed in detail in Section 4), 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 of treating all input types.

Finally, this study investigates the ROI (Return on Investment) for addressing performance issues. 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. This analysis helps practitioners choose between localized or design-level optimization solutions, and prioritize issues due to different root causes. We found that design-level optimization requires more developers and more discussions compared to localized optimization; but it does not warrant higher performance improvement. However, we argue that design-level optimization will provide long-term benefits in other aspects, such as design quality and code maintainability. The tricky part is that these benefits are often not explicit to measure and not immediately visible to practitioners. Consequently, practitioners are more likely to choose localized optimization over design-level optimization, for convenience and immediate benefits. We conjecture that this is how technical debts start to accumulate [30].

In summary, the key contributions of this study are:

1) This study reveals eight common root causes and the corresponding resolutions to performance issues based on 192 real-life performance issues. This study surveyed 60 related literature that investigated these root causes and found 24 available tools that can detect and/or fix different performance issues.

2) This study provides empirical findings of design-level optimization that are necessary for addressing some performance issues. This study revealed four typical design-level optimization patterns. We believe that these findings are the first of their kind.

3) This study measures the Return on Investment (ROI) for addressing performance issues. It compares the ROI of localized and design-level optimization resolutions, and compared the ROI of performance issues due to different root causes.

4) This paper contributes a new design structure modeling technique for analyzing design-level optimization, named *Diff-Design Structure Matrix*. This technique can be used for research related to software design structure evolution.

5) This study contributes a rich, high-quality dataset of 192 performance issues, with the annotated information regarding: the symptoms, the root cause, the resolution, and the profiling data. The data is available here: https://sites.google.com/view/icpe-archperf-2020/home.

2 RESEARCH QUESTIONS

This paper aims to answer three research questions.

RQ1: What are the common root causes of real-life software performance issues? Is each type well-addressed in the existing literature? 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: 1) **what are the common root causes of performance issues (RQ-1.1)?** We assume that there are common root causes that recur in different software projects. And 2) **how well is each root cause addressed in the literature (RQ-1.2)?** We are particularly interested in tools that detect and fix different performance issues.

RQ2: Are performance issues addressed by design-level optimization? 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 three parts: 1) **are performance issues usually addressed by localized optimization or complicated design-level optimization (RQ-2.1)?** We analyze the scope of the performance resolution and examine the design structure change to answer this question. 2) **What are the typical design-level optimization patterns (RQ-2.2)?** If some performance issues require design-level resolution, we further investigate what are the typical design-level optimization patterns and why they are necessary. And 3) **how prevalent is each design-level optimization pattern, especially for addressing different root causes (RQ-2.3)?** This part investigates the application of different design-level optimization patterns for issues with different root causes.

Table 1: Study Subjects

Subject	Since	#Issues	Perf-key	Verified	Solved
PDFBox	2008	3855	135	93	74
Avro	2009	2151	135	113	41
Ivy	2005	1522	54	41	18
Commons-Collections	2006	435	51	46	23
Groovy	2003	8476	137	107	36
Total		16439	512	400	192

RQ3: What is the ROI (Return on Investment) for fixing performance issues? Software development is constrained by limited resource and time. This RQ helps practitioners treat performance issues economically. Our investigation is three-fold: 1) **what is the overall ROI for addressing performance issues (RQ-3.1)?** 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". 2) **How is the ROI of localized and design-level optimization compared to each other (RQ-3.2)?** The purpose is to compare the ROI of localized and design-level optimization to help prioritize different optimization strategy. And 3) **how is the ROI of performance issues affected by different root causes (RQ-3.3)?** This question aims to provide insight for the practitioners to prioritize performance issues of different root causes.

3 STUDY SUBJECTS AND APPROACH

3.1 Study Subjects

We study performance issues from five Apache open source projects: PDFBox[31], Avro[32], Ivy[33], Commons-Collections[34], and Groovy[35] as listed in Table 1. The PDFbox library is a Java tool for working with PDF documents. Avro is a remote procedure call and data serialization framework. Ivy is a transitive package manager to resolve complex project dependencies. Commons-Collections is a Java collections library of JDK Collection, Set, List and Map interfaces. Groovy is a Java-syntax-compatible object-oriented programming language for the Java platform.

These subjects were selected due to the following considerations. First, they are in different domains, such as document processing, data serialization, web server etc. Performance plays a critical role in all these projects. The goal is to draw general observations across different problem domains. Second, these projects are all well accepted, successful, and are all still active Apache open source projects. The source code, version control repository, and bug-tracking systems are all well organized and readily available. This provides high quality data for our study.

3.2 Study Approach

Figure 1 shows the overview of this study with four steps:

- *Step 1: Data Collection.* This step collects performance issues that are resolved from the five projects.
- *Step 2: Issue Annotation and Categorization.* This step manually annotates/summarizes key information, including the symptom, root cause, proposed resolution, and profiling data, in each issue report and the respective code revisions. We categorize the collected issues based on their root causes and corresponding resolutions. This step addresses RQ1.

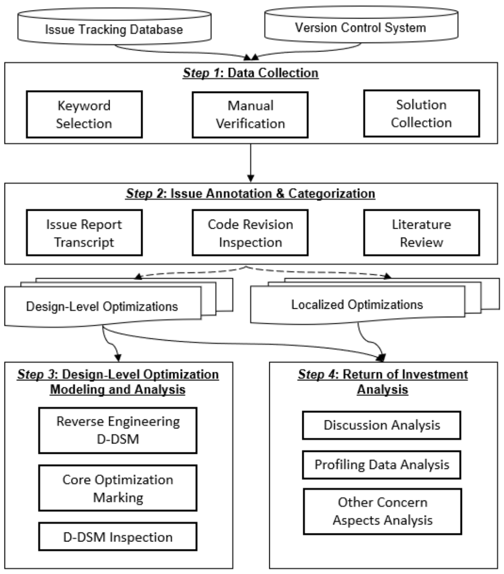


Figure 1: Study Overview

- *Step 3: Design-level Optimization Modeling and Analysis.* This step models and analyzes the design-level optimization, where a group of source files are revised simultaneously for addressing a performance issue. This step contributes a new modeling technique named *Diff-Design Structure Matrix* to facilitate the analysis. This step addresses RQ2.
- *Step 4: Return on Investment Analysis.* This step analyzes 1) the investment—in terms of the number of engaged developers and the number of discussions, for addressing each performance issue, as recorded in the JIRA bug tracking system, and 2) the return—the extent of performance improvement based on available profiling data. This step addresses RQ3.

Step 1: Data Collection. As shown in Table 1, we initially collected a total of 16439 issues (Column 3) from the JIRA bug tracking database[36], dated back to the creation of each project. Next, we apply keyword matching to select issues relevant to performance, similar to the practice in prior studies [9, 37, 38]. 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"*, which combine the keywords used in previous 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 512 issues were kept by matching relevant keywords. Over a six-week long effort, we manually verify each issue to exclude false positives. For example, "performance" sometimes refers to the productivity of the developers. Two authors separately inspect the issues matched by keywords. The authors drop uncertain issues to ensure the quality of the data set. This further distills a total of 400 issues shown in Column 5 of Table 1. Finally, we retrieve the code revisions that address each performance issue. The code revisions can be extracted from the version control system by locating issue IDs that appeared in the commit messages [39]. Issues without linked solutions, either because they were not solved or because the linkage was missing, are dropped. Thus, we finally identified 192 resolved performance issues.

Table 2: Data Annotation Statistics

Subject	Symptom	Cause	Solution	Profiling	Other
PDFBox	89%	96%	99%	41%	12%
Avro	63%	73%	100%	61%	7%
Ivy	89%	83%	94%	17%	17%
Commons-Collections	96%	91%	100%	57%	0%
Groovy	92%	97%	94%	22%	3%
Total	85%	90%	98%	41%	8%

Step 2: Issue Annotation and Categorization. We manually inspect and annotate five key aspects of information (if available) in each issue report. They are: 1) the symptoms, 2) the root cause, 3) the proposed solution, 4) the profiling data, and 5) any other aspects of concerns (e.g. maintainability issues). Figure 2 is an example of an annotated issue report, *PDFBOX-591*[40]. Table 2 shows the statistics of the annotation: There are 85% of issues that described the symptoms of the issues. 90% and 98% of the issues contain discussions about the root causes and solutions. And 41% of the issues contain profiling data. Only 8% performance issues contain discussions regarding other aspects of concerns, such as maintainability and design.

PDFBox-591: PDFBox Performance Issue: BaseParser.readUntilEndStream() rewrite
"**Symptom:** The load time for loading documents into PDFBox (PDDocument) is too slow.
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."

Figure 2: Issue Annotation-PDFBOX-591

Next, we manually review and summarize the code revisions that fixed the issues. The code revisions reveal the most essential logic of the root causes and solutions to performance issues. Based on the annotation and summary of code revision, we apply open coding to discover recurring types of performance issues in different projects, following the grounded theory methodology introduced in [41].

We perform extensive literature review to evaluate how well each root cause is addressed in existing literature. First, we used "performance" as the keyword to search papers on Google Scholar, following [42]. We screened the titles and abstract of the top 500 papers, and we found 47 papers that are relevant to detecting and solving real-life performance issues. Next, we searched the references in the *Related Work* and/or *Discussion* sections of the 47 papers, following the "Backward Snowballing" methodology [43]. We found another 45 related papers. We carefully reviewed the 92 (47+45) papers. Among these, 60 papers investigated one or more root causes of performance issues. The other papers are empirical studies, general performance modeling, profiling techniques, and others work that is not directly related to addressing the identified root causes.

Step 3: Design-level Optimization Modeling and Analysis. This paper contributes a new modeling technique, called *Diff DSM (D-DSM)*, to capture the essential design structure change in a complicated code revision.

The *D-DSM* is built upon Baldwin and Clark [44]'s *Design Structure Matrix (DSM)*. The original DSM is a square matrix with its rows and columns labeled by the same set of design element names and/or numbers, in the same order. A cell along the diagonal represents self-dependency, and an non-empty off-diagonal cell captures the dependency between the element on the row to the element on the column. *DSM* is often used in modeling software systems [45, 46]. The elements can represent source files. Each cell captures the different types of structural dependencies from 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. Figure 4b is an example showing the dependencies among 4 source files from Apache project, PDFBox. The rows and columns represent source files, arranged in the same order. The cells display the dependencies among these files. Cell[3,1] says "dp", meaning *PDFStreamEngine* (row 3) has a structural dependency to *Matrix* (row 1).

The *D-DSM* is built upon the *DSM* but with the following uniqueness: 1) it only contains the changed files and their structural dependencies in a revision; 2) it highlights the added/removed files; and 3) it highlights the added/removed structural dependencies among involved source files. A *D-DSM* is automatically computed by taking a revision ID and the project repository as the input. The computation is accomplished in three main steps as shown in Figure 3. First, we use a Git command to generate two versions of the code base: one reverted to before the specified revision; the other reverted to after the revision. Next, we use existing reverse engineering tools to recover the structural dependencies among source files and generate two DSM files to represent the dependencies before and after the revision. Finally, we calculate a D-DSM by comparing the two DSM and highlight the added/removed source files and the changes to the dependencies among files.

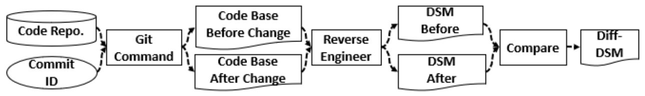
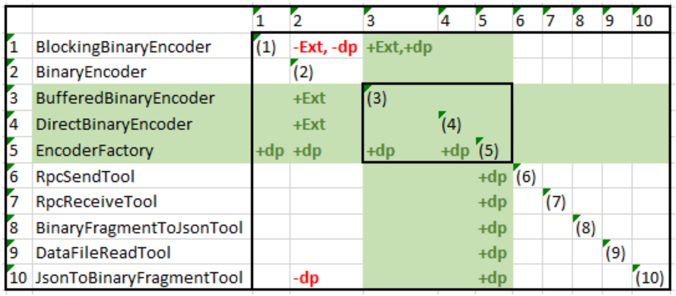


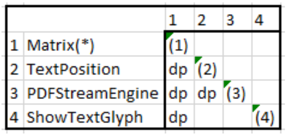
Figure 3: D-DSM Modeling Overview

To reveal the patterns in design-level optimization, we separate code revisions into two groups based on the scope of change: 1) *localized optimization* that revises a single source file; and 2) potential *design-level optimization* that simultaneously revises a group of source files. Admittedly, simultaneously revising a group of source files does not always imply a design-level optimization. Developers may combine multiple change requests, e.g. fixing a bug, with performance optimization. We manually verify and exclude issues where a group of source files are revised together for non-performance related reasons. For example, in issue *PDFBOX-1924*[47], the main purpose is fixing a functional bug. Developers revised four source files—only one line of code is for fixing the performance issue. The resolution to *PDFBOX-1924* is considered a localized optimization. For each complicated performance issue resolution, we use *D-DSM* to formally and automatically capture the essential design structure change in it. This helps us to reveal typical patterns in the design-level optimization.



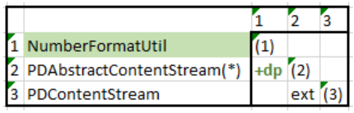
		1	2	3	4	5	6	7	8	9	10
1	BlockingBinaryEncoder	(1)	-Ext, -dp	+Ext,+dp							
2	BinaryEncoder		(2)								
3	BufferedBinaryEncoder		+Ext	(3)							
4	DirectBinaryEncoder		+Ext		(4)						
5	EncoderFactory	+dp	+dp	+dp	+dp	(5)					
6	RpcSendTool					+dp	(6)				
7	RpcReceiveTool					+dp		(7)			
8	BinaryFragmentToJsonTool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

(a) Classic Design Pattern: Avro-753



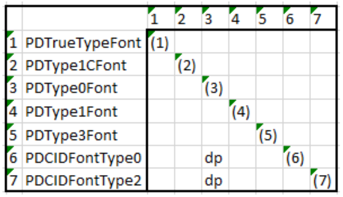
		1	2	3	4
1	Matrix(*)	(1)			
2	TextPosition	dp	(2)		
3	PDFStreamEngine	dp	dp	(3)	
4	ShowTextGlyph	dp			(4)

(b) Type I Propagation: PDFBox-893



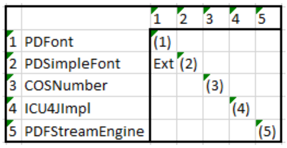
		1	2	3
1	NumberFormatUtil	(1)		
2	PDAbstractContentStream(*)	+dp	(2)	
3	PDContentStream		ext	(3)

(c) Type II 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.

Figure 4: Design-level Optimization Patterns

As an illustrative example, Figure 4a is the *D-DSM* of the resolution to issue *AVRO-753* [48]. Due to space limitations, this figure only shows the ten most important of the 25 revised source files. The rows and columns represent the ten changed files, ranked in the same order. The newly added files (row 3 to row 4) are highlighted in shaded background. The cells represent the dependencies among files. For example, entry[5,1] says "+dp", which indicates that *EncoderFactory* (row 5) *depends* on *BlockingBinaryEncoder* (row 1). The symbol "+" means that it is a newly added dependency. Similarly "-" means the following dependency is removed in this revision. In Figure 4a, we can observe the introduction of a factory design pattern for addressing this issue. The three added files are the key factory pattern elements, and the newly introduced dependencies

Table 3: Inefficient Data Structure and Replacement

Inefficient Data Structure	Replacement	# of Issues
Array, List	Set, Map	14
String, StringBuffer	StringBuilder	6
HashMap, WeakHashMap	ConcurrentHashMap	3
Integer, Float, Double	int, float, double	2
LinkedList	ArrayList	1
HashMap	TreeMap	1
Others		6

are all associated with reference to the *EncoderFactory*. In this case, the newly added files implements a factory pattern, where each concrete pattern provides a solution that optimizes performance in a different situation. We will discuss this case in detail in Section 4.

Step 4: Return on Investment Analysis. In this step, we examine the ROI of resolving performance issues. As the proxy of the invested effort, we measure two aspects:

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

Among the 192 issues, only 76 contain profiling data that provide performance metrics of before and after the issue resolution. We found that most issues used response time (where the time unit used could be different), and only two issues used throughput. To avoid confusion, we define a unified **Improvement Factor** to measure the extent of improvement by 1) $\frac{ResponseTime\_BeforeFix}{ResponseTime\_AfterFix}$; or 2) $\frac{Throughput\_AfterFix}{Throughput\_BeforeFix}$, depending on the used metric.

4 STUDY RESULTS

4.1 Common Root Causes

RQ-1.1 What are the common root causes to real-life performance issues? We observed **eight** types of root causes that recur in the 192 performance issues. Each root cause has 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. Table 3 shows the common replacement patterns. For example, the most common (14 out of 36) case is to replace *Array* or *List* by *Set* or *Map*. This makes data searching faster. In 6 cases, developers replaced *StringBuffer* by *StringBuilder*, since the latter creates new *String* more efficiently. Such empirical experience helps practitioners prevent inefficient data structure.

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 reuse [10]; and 2) only perform the computation when the input status changes.

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 [37, 49–52]. Typical resolution is to 1) add checking conditions for the special cases, and 2) employ special algorithms to treat each special case efficiently.

Inefficient Iteration (II): The status of loop iterations remains the same and the iterations become useless. Typical resolution is to check whether the loop status becomes stable; and, 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 problems are caused by sub-optimal choice of APIs [53]. Typical resolution is to choose or re-implement an efficient API [54–56]. For example, in addressing *GROOVY-7977* [57], the LRUCache was replaced by Caffeine, a high performance caching library for Java 8. There is a diverse amount of API replacements, and we did not observe prevalent patterns.

Redundant Data Processing (RDP): These performance issues are caused by redundant or tedious data processing. It usually involves copying or processing a large chunk of data in small units, such as bit by bit or pixel by pixel. The typical reresolution is to copy or process the data in one go. For example, in issue *AVRO-556* [58], the developers originally *"read bytes into the result vector one-byte-at-a-time"*, which, according to the developers, *"is horrendously slow"*. The fix is to copy all the bytes in a single call.

Multi-threaded Blocking (MTB): 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 have to wait for each other. In the worst cases, different threads may even got blocked, resulting in lengthy execution/waiting time. The resolution is usually to improve the synchronization mechanism.

General Inefficient Computation (GIC): These performance issues are caused by general inefficient computation. They are usually addressed by algorithmic improvements. As an example, in issue *PDFBOX-600* [59], the order of two checking conditions in a *and* operator caused unnecessary computation since the first checking condition is true and the second checking condition is false most of the time. The developers switched the order to avoid checking both conditions.

Prevalence of Root Causes: As shown in Figure 5, there are four prevalent root causes, each accounts for about 20% of performance issues in our dataset. They are: *General Inefficient Computation*, *Inefficiency under Special Cases*, and *Repeated Computation*, and *Inefficient Data Structure*. In comparison, in Selakovic et al.'s study, they found that API-related root cause is the most prevalent, accounting for 52% of the studied issues [28]. It is only responsible for 8% of the issues in our dataset. In Jin, et al.'s work, almost half of the performance issues are related to *Inefficiency under Special Cases* [37]. We believe that the difference is due to at least two factors. First, the project domain. Selakovic et al. mostly studied projects related to framework or library. While, our study subjects are in a variety of problem domains. Thus the percentage reported in this study is likely to be more representative of general performance issues. Second, the programming languages. Selakovic et al. studies exclusively JavaScript projects. The issues studied in this paper are mostly Java. Particularly, *Multi-threaded Blocking* was not discussed in Selakovic, et al.'s study, since JavaScript does not support multi-threaded programming.

Table 4: Available Tools

Root Cause	Tool	Language	Year(A.)
Inefficient Data Structure	[D,F]:Perflint [16]	C++	2009(A)
	[D,F] CoCo [17]	Java	2013
	[D,F]: CHAMELEON [25]	Java	2009
	[F]: Brainy [26]	C++	2011
	[D,F]: CollectionSwitch [60]	Java	2018
Repeated Computation	[D]: Cachetor [21]	Java	2013(A)
	[D,F]: MemoizeIt [10]	Java	2015(A)
			2015
Inefficiency under Special Cases	[D]: PerfFuzz [50]	C	2018
	[D]: GA-Prof [49]	Java	2015
Inefficient Iteration	[D,F]: Caramel [23]	Java/C/C++	2015
	[D]: Toddler [12]	Java	2013(A)
	[D]: GLIDER [11]	Java	2016(A)
	[D]: LDoctor [61]	Java/C/C++	2017
	[F]: Clarity [13]	Java	2015
Inefficient API Usage	[D,F]: BIKER [54]	Java	2018
Redundant Data Processing	[F]: RowClone [62]	*assembly*	2013
	[F]: LazyClone [63]	Java	2015
Multi-threaded Blocking	[D]: SpeedGun [9]	Java	2014
	[D,F]: SyncProf [18]	C/C++	2016
	[D]: LIME [64]	C/C++	2011
	[D,F]: SHERIFF [65]	C/C++	2011(A)
	[D]: PRADATOR [66]	C/C++	2014(A)
General Inefficient Computation	[D]: Trend Profiler [67]	C	2007
	[D]: Spectroscope [68]	Perl/C++	2011(A)
	[D]: PerfPlotter [69]	Java	2016(A)

Note: "D" means the tool can automatically **detect**.
"F" means the tool can automatically provide **fixing** resolutions.

RQ-1.1 Implication: Practitioners should be aware of the common root causes that recur in different projects when they fix performance issues. This awareness also helps practitioners to prevent performance issues in software design and development, instead of treating performance as an afterthought.

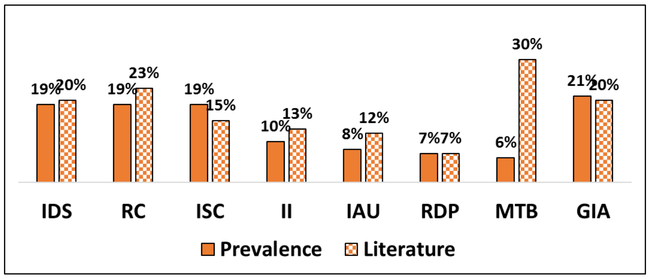


Figure 5: Prevalence and Literature of Different Root Causes

RQ-1.2: How well is each root cause addressed in literature? We found that 60 relevant papers published between 2000 to 2019 from ICSE (20%), PLDI (13%), OOPSLA (10%), FSE (8%), ICPE (8%), ISSTA (7%), ECOOP (5%), and other conferences or journals (28%). Section 6 discusses the details in existing literature. The distribution of focuses is shown in Figure 5. The three most frequently studied root causes are: *Multi-threaded Blocking* (30%), and *Repeated Computation* (23%), *Inefficient Data Structure* (20%), and *General Inefficient Computation* (20%). Table 4 lists the 24 tools to detect and/or fix performance issues in the related literature. Most of them can detect the issues, but only 50% can automatically fix the issues. The majority of the tools can be applied to Java (60%) or C/C++ (50%) projects. Only one is applicable to Perl projects. Only 9 of them provided public accessible links, dated back to 2009.

RQ-1.2 Implication: Practitioners may benefit from existing tools when facing similar issues. However, there are several potential concerns: 1) The proposed tools have not been tested and compared to each other on any benchmark dataset; 2) Tools are limited to Java/C/C++ projects; and 3) The availability and usability of these tools are potential obstacles for practitioners to using them.

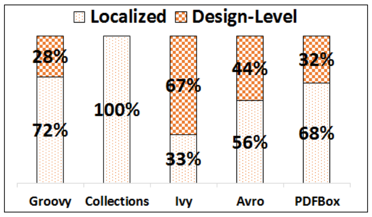
4.2 Design-level Optimization

RQ-2.1 Are performance issues usually addressed by localized optimization or complicated design-level optimization? In general, the majority (67%) of performance issues are fixed by localized code revisions. The remaining 33% are addressed by design-level optimization. Figure 6a shows the distribution of localized and design-level optimization in each project. The performance issues in Apache Commons-Collections are exclusive addressed by localized optimization. In the other four projects, from 28% (in Groovy) to 67% (in Ivy) of performance issues require design-level optimization. Figure 6b shows the distribution in each type of root cause. *Inefficient Iterations* are exclusively addressed by localized optimization. The other types all require non-trivial amounts of design-level optimization: from 22% to 67%.

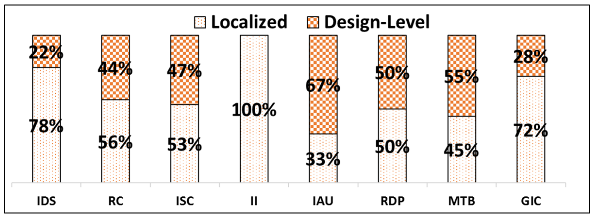
RQ-2.1 Implication: Practitioners should be aware of the need for design-level optimization. This need can be impacted by the nature of projects, as well as the nature of the root causes. For example, Collections is a simple JDK collection library and does not deal with complicated interactions among domain elements. Thus, its issues are exclusively resolved by localized optimization. In comparison, the other projects are from more complicated domain, such as document processing. As such, they require non-trivial design-level optimization. In addition, *Inefficient Iterations*, by their nature, are local to "for loops", as such, they are resolved exclusively by localized optimization.

RQ-2.2 What are the typical design-level optimization patterns? With the help of the *D-DSM*, we revealed four patterns:

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*[48] is caused by *Inefficiency under Special Cases*. The *BinaryEncoder* is really slow when processing data chunks smaller than 128 bytes. The factory pattern provides an elegant design for treating different input cases in separate. Figure 4a 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) *EncoderFactory* (row 5), which is the factory pattern interface; 2) *BufferedBinaryEncoder* (row 3), a concrete encoder that efficiently deals with large data chunk by using a buffer; and 3) *DirectBinaryEncoder* (row 4), the other type of encoder that efficiently deals with small data chunk without buffer. 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



(a) By Project



(b) By Root Cause

Figure 6: Localized vs. Design-Level Optimization

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 propagation: Type I: The optimization core propagates changes to a group of source files that structurally depend on and benefit from the core. For example, Figure 4b is for issue *PDFBOX-893*[70]. 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 II: The optimization core propagates changes to a group of source files that the core depends on, to support the core. For example, Figure 4c is a Type II propagation for issue *PDFBOX-3421*[71]. The optimization core is *PDAbstractContentStream*, which suffers from 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*[72] is such an example, shown in Figure 4d. 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*[73], 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 belongs to the “text extraction" component.

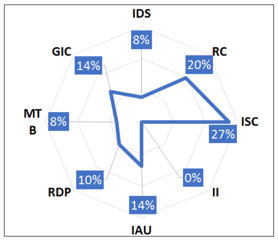
RQ-2.2 Implication: According to Smith and Williams [2], most performance issues have their roots in poor architectural decisions made before coding is done. Our results on these four patterns reinforce this argument. They represent four design strategies to resolve performance issues. For example, the issue in Figure 4a, if treated by a localized optimization, will result in a god class that treats all different input types. Meanwhile, practitioners should pay attention to *change propagation* when implementing an optimization. Code clone is a notorious code smell [74]. *Optimization clone* indicates that practitioners should at least be aware of code clones for thorough optimization, if not fixing the clone by refactoring. *Parallel Optimization* suggests that practitioners should seek architecturally related opportunities in performance optimization.

RQ-2.3 How prevalent is each design structure pattern, especially for addressing different root causes? The majority of design-level optimization are *Change Propagation*: 41% in Type I and 27% for Type II. Each of the other three patterns accounts for about 10%. Figure 7 show the application of the four patterns for addressing different root causes in four radar charts. In a particular note, *Inefficient Iterations* are excluded in this discussion, since they are exclusively addressed by localized optimization. Figure 7a shows that *Change Propagation* applies for addressing **all** different types of root causes. According to Figure 7b, *Optimization Clone* is **not** applied for addressing *Inefficiency under Special Cases*. Figure 7c shows that *Classic Design Patterns* are **not** applied for addressing *Inefficient Data Structure* and *General Inefficient Computation*. But almost **half (43%)** are applied for addressing *Inefficiency under Special Case*. Figure 7d shows that the *Parallel Optimization* is **not** applied for addressing *Inefficiency under Special Cases* or *Inefficient API Usage*.

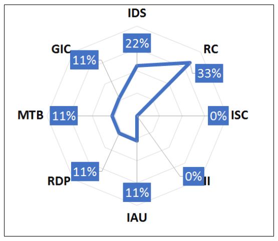
RQ-2.3 Implication: Overall, the applications of four patterns on addressing different root causes are quite different from each other. Practitioners should be aware of the different prevalence of patterns in both software design and development.

4.3 Return on Investment

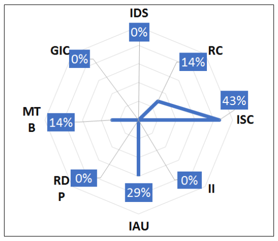
RQ-3.1 What is the overall ROI for addressing performance issues? The analysis result is shown in Figure 8. According to Figure 8a, the majority 58% of performance issues are addressed by 1 or 2 developers. According to Figure 8b, 64% of issues are addressed with no more than 5 discussion comments. Figure 8c shows that 66% of the issues are fixed with less than 10 Improvement Factor.



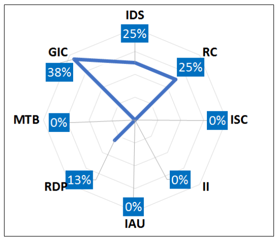
(a) Change Propagation



(b) Optimization Clone

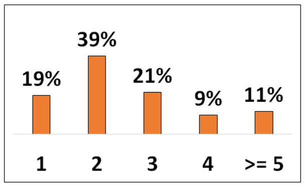


(c) Design Pattern

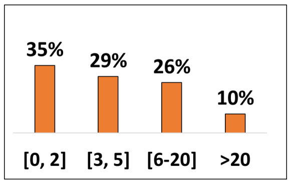


(d) Parallel Optimization

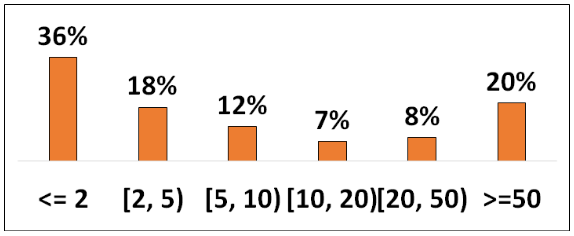
Figure 7: Design-Level Patterns for Different Root Causes



(a) # Developers

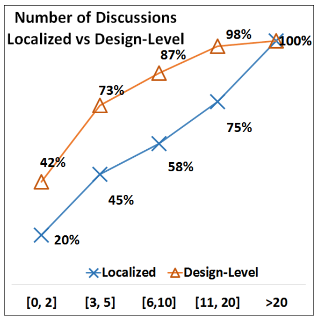


(b) # Discussions

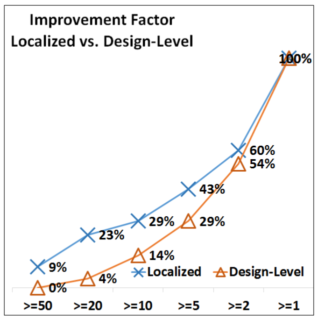


(c) Improvement

Figure 8: ROI for All Performance Issues



(a) # Discussions



(b) Improvement Factor

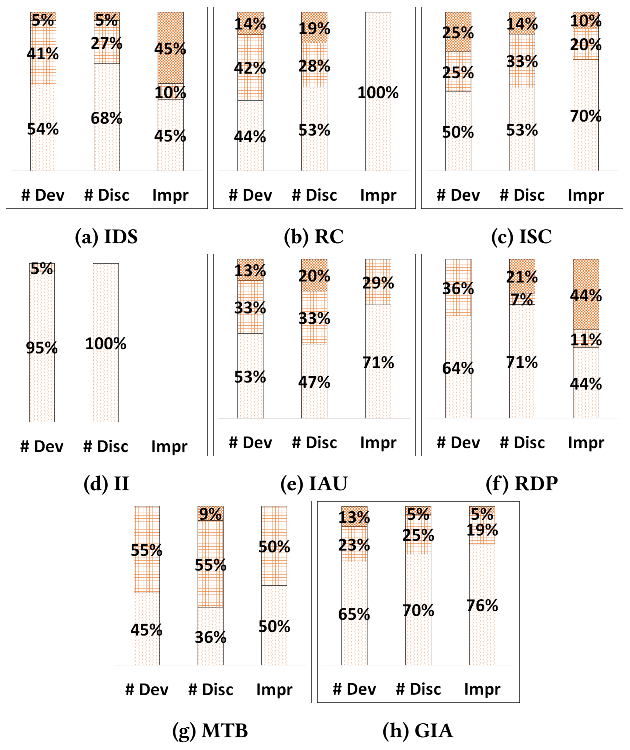
Figure 9: ROI for Localized vs. Design-Level Optimization

RQ-3.2 How is the ROI of design-level optimization compared to localized optimization? The result is shown in Figure 9. Figure 9a is the cumulative distribution function plot of the number of discussions for localized (the line with the cross marker) and design-level (the line with the triangle marker) optimization. The x-axis are ranked from low to high in the number of discussions. The plot of the localized optimization is constantly above the plot of the design-level optimization. This means that a larger portion of localized optimization requires less discussions compared to the design-level optimization. For example, Figure 9a indicates that 42% of localized issues, while only 20% of design-level issues, require no more than 2 discussion comments. The average number of discussions on localized optimization is 4.82 and the average on design-level optimization is 15.23. We performed the same analysis for the number of engaged developers, the details of which are not included here due to space limit. However, we made consistent observation: localized optimization usually require less number of developers compared to design-level optimization. Thus, we conclude that the investment on localized optimization is obviously less than the design-level optimization.

Figure 9b is the cumulative distribution function plot of Improvement Factor of the localized (the line with the cross marker) and design-level (the line with the triangle marker) optimization. The x-axis is ranked from large to small improvement factor. As such, the plot on the top indicates higher improvement in general. For example, Figure 9b indicates that 9% of localized optimization achieved more than 50 times performance improvement; but none of the design-level optimization were able to achieve this much improvement. The plot of the localized optimization is slightly higher than that of the design-level optimization. The median of localized and design-level optimization is 6.9 and 2.5 respectively. Therefore, we conclude that localized optimization offers higher performance improvement compared to the design-level optimization.

RQ-3.2 Implication: Design-level optimization is more difficult to develop compared to localized optimization; but it does not warrant higher performance improvement. However, design-level optimization may provide benefits other than performance improvement. For example, we observed that 15 issues discussed other aspects of concerns, such as code readability and maintainability—73% of these issues employed design-level optimization. The tricky part is that these benefits are not explicitly measurable or immediately visible to practitioners. As such, practitioners tend to favor localized optimization for convenience and immediate benefits.

RQ-3.3 How is the ROI of performance issues affected by different root causes? Figure 10 shows the ROI for each root cause. In each sub-figure, the three vertical bars show the distribution of the number of developers (#Dev), the number of discussions (#Disc), and the Improvement Factor (Impr) for performance issues caused by each root cause. As shown in the legend on the bottom of Figure 10, each vertical bar is divided into three levels: 1) For the number of developers, we separate [1,2], [3,4], or >=5 developers; 2) For the number of discussions, we separate [0,5], [6,20], or >= 21 discussions; 3) For the Improvement Factor, we separate [1,10),[10,50), >= 50.



Legend	# Developers	# Discussions	Improvement Factor
▫	[1, 2]	[0, 5]	[1x, 10x]
▦	[3, 4]	[6, 20]	[11x, 50x]
▨	>= 5	> 20	> 50x

Figure 10: ROI for Different Root Causes

We can make the following key observations: 1) *Inefficient Iterations* (Figure 10d) require the **lowest** cost to address: 95% of these issues are addressed by 1 or 2 developers, and 100% of these issues require at most 5 discussions. However, we didn't find any performance profiling data available for this type. 2) *Inefficient Data Structure* (Figure 10a) and *Redundant Data Processing* (Figure 10f) have very high ROI. For example, the majority (54% and 68%) of *Inefficient Data Structure* issues require 1 or 2 developers, and within 5 discussions. However, 45% of them achieved more than 50 times performance improvement. 3) *Repeated Computation* (Figure 10b) tend to have the **lowest** ROI. All of these issues have improvement factor of less than 10. However, none-trivial (14% and 19%) of these issues require **high**—more than 5 developers and more than 20 discussions—to address.

RQ-3.3 Implication: Fixing issues caused by different root causes can provide very different ROI. Practitioners should be aware of this when prioritizing issues due to different causes. However, we acknowledge that the ROI may be impacted by many other factors. We plan to further pursue this in future work.

5 LIMITATIONS AND VALIDITIES

Limitations. First, we did not evaluate the actual effectiveness and usability of the 24 tools for detecting and/or fixing performance issues. In future work, we plan to collect and use these tools on our dataset. Second, the performance improvement is evaluated based on the available profiling data contained in the issue reports. We did not execute the code and compare the performance metrics before and after the issues were fix. Therefore, we only have the performance improvement of 76 issues with available profiling data. In future work, we will try to evaluate the improvement of all the 192 issues by executing the code. Third, we evaluated the ROI of performance issues only based on three measurements: the number of engaged developers, the number of discussions, and the performance improvement. We acknowledge that there are other meaningful measurements of effort and benefits. For example, we did not measure effort in terms of the amount of time needed. In addition, we were not able to measure benefits in terms of design improvement or maintenance quality. We also acknowledge that the ROI is based on the investment on resolving the issues, without considering the effort needed for discovering these issues. Lastly, we acknowledge that different programming languages can impact the presented results. The issues studied in this work are mostly in Java. In future work, we will investigate the impact of language on performance issues and their ROI.

Threats to Validity. First, this study is based on 192 real-life performance issues. We cannot guarantee that we have captured all possible types of performance issues. We acknowledge that the choice of keywords for matching performance issues could have an effect the results presented. And, we also cannot guarantee that the same conclusions still hold for different performance issue dataset. For example, we found that Apache Commons Collections does not contain any design-level optimization. Therefore, the statistics reported in this paper may vary based on project domains, programming languages, and other factors. However, we argue that any empirical study will suffer from this threat to validity. We plan to extend our study and test these findings more extensively. Meanwhile, we have shared the data of this study for replicating studies. Second, we acknowledge that the analysis of common root causes and solutions to performance issues is potentially biased by the authors' understanding and experience. In many cases, the boundary between different root causes could be blurry. For example, an inefficient API may has its root cause in an inefficient data structure. Such cases are considered as both types. This is an internal threat to validity. To best avoid personal basis, we have different authors work together, and we performed an extensive literature review. We confirmed the different types of performance issues discovered in this study are consistent with the findings of previous studies.

6 RELATED WORK

Inefficient Data Structure has been well-studied in prior works [5, 8, 16, 17, 25, 26, 28, 60, 75–79]. Costa et al. found that data structures have major impact on software performance [60, 79]. The detecting and fixing approaches mostly rely on runtime profiling [78]. All the five tools, Perflint [16], Coco [17], Brainy [26], CollectionSwitch [60], and Chameleon [25], monitor dynamic execution to recommend potential replacements. In addition, Xu el al. proposed a static analysis to identify inefficient data structures [75]. Other prior work focuses on a specific scenario of inefficient data structure [8, 76, 77]. For example, Hunt et al. studied the relationship between speed and energy consumption of various lock-free data structures [77].

Repeated Computation has been studied in [5, 6, 10, 28, 80, 81, 81–87]. The general solution is to add cache or buffer to store calculated results [6]. The two tools, Cachetor [21] and MemoizeIt [10], automatically detect opportunities to cache calculated results by comparing the inputs and outputs of method calls, based on dynamic analysis. Memory consumption is a great concern for most software systems. Infante proposed a technique to identify opportunities to reduce memory consumption by optimizing caches [86]. Other prior works studied various strategies of cache optimizations [80, 81, 81, 87]. For example, Li et al. attempted to determine the optimal caching decisions across the network in order to minimize average latency [87].

Inefficiency under Special Cases have been studied in [4, 28, 37, 49–52, 82, 85]. Available approaches highly reply on available testing inputs to reveal inefficiency [37, 52]. Coppa et al. focused on the size of input [51]—they measure how the performance of routines scales with the input size. Shen et al. proposed GA-Prof that uses highly structured inputs [49] to accurately detect performance bottlenecks. This approach encodes highly-structured inputs as genes by using an genetic algorithm. In contrast, PerfFuzz requires no domain knowledge since its inputs are represented as byte sequences [50].

Inefficient Iterations have detecting/fixing tools based on both static and dynamic analysis [11, 12, 23, 28, 37, 61, 85]. Caramel is a static analysis tool that detects inefficient iterations by adding conditional-breaks [23]. Clarity is also a static analysis tool, focusing on detecting nested loop traversals [13]. Toddler, in contrast, dynamically detects inefficient iterations by finding repetitive memory accesses [12]. 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 [11]. The limitation of static analysis tools are that they can only find a subset of inefficient iterations; while the limitation of dynamic analysis tools is that they will slowdown the program. Song et al. [61] proposed a static-dynamic hybrid analysis tool, LDoctor, which is faster than than Toddler and more effective than Caramel.

Inefficient API Usage has been studied in [28, 37, 53–56, 88, 89]. Selecting which third-party libraries to use is highly dependent on programmers knowledge and experience [55]. Kawrykow et al. found that inefficient API Usage is quite common [53, 56]. They proposed a static analysis approach to replace code by available APIs that provides similar functions. Well-accepted APIs are usually more efficient [56]. The limitation is that the APIs have to be already used in the targeted software application. Huang et al. proposed BIKER, an API recommendation approach, that leverages Stack Overflow posts to recommend and prioritize candidate APIs for a program task.

Redundant Data Processing has been studied in [28, 62, 63, 90–92]. Research found that processing data in large chunks is much faster than processing them unit by unit is loops [62]. Chen et al. proposed an automated approach to detect redundant data processing specific to database [90]. Two well-known approaches to process data in chunks are the shallow clone and the deep clone [91]. Shallow clone only clones the main object without their dependencies, which compromises the information. The deep clone copies the entire dependency graph, which is time and memory consuming. Cartaxo et al. proposed an intelligent cloning approach, Lazy Clone [63]. Seshadri et al. improve data cloning at the hardware level [62].

Multi-threaded Blocking has been studied in [5, 9, 18–20, 37, 64–66, 82, 84, 88, 92–98]. SpeedGun generates multi-threaded performance test cases to expose performance difference between two program versions [9]. SyncProf can use these performance test cases to detect and optimize sychronization bottlenecks [18]. LIME and PRADATOR also rely on the availability of test cases [64, 66]. LIME focuses on load imbalance between threads. PRADATOR and SHERIFF focus on false sharing problem of objects, and SHERIFF has higher accuracy compared to PRADATOR [65].

General Inefficient Computation has been studied in [5, 28, 37, 67–69, 82, 84, 85, 90]. The algorithms in a program have the most basic influence on software performance [67]. The challenge to address this type of issue is that the complexity derived from mathematical analysis cannot precisely reflect the runtime complexity [69]. Thus, Goldsmith et al. proposed Trend Profiler to measure the run-time complexity by executing a program on workloads spanning several orders of magnitude [67]. Spectroscope [68] also uses dynamic profiling to detect hot-spots in running programs. The problem is that the accuracy of profiling relies on the given set of test inputs. Chen et al. proposed PerfPlotter, which can accurately capture the best and worst cases and the distribution of program execution times [90].

There are also other empirical studies that investigated the categorization of performance issues [28, 37, 84]. Jin et al. categorized the root cause of 109 real-world performance bugs into four types, which are all included in this study. Liu et al. focused on performance issues from Android smart-phone applications [84]. They observed three types of performance issues: 1) GUI lagging, 2) energy leak, and 3) memory bloat [84]. Selakovic et al. summarized seven types of root causes from JavaScript projects [28]. Multi-threaded blocking, identified in this paper, does not apply to JavaScript projects.

7 CONCLUSION

This paper investigated 192 real-life performance issues, and identified eight recurring root causes and typical resolutions. There are existing techniques and tools for detecting and fixing these root causes. However, the actual effectiveness and usability of these tools have not been evaluated and compared to each other using the same benchmark dataset. This calls for more research in the future. We found that developers resolved 33% of the 192 issues through design-level optimization, manifested in four different patterns. This finding reinforces the view that performance issues can be rooted in bad software design decisions. In the ROI analysis, we found that localized optimization provides higher ROI than design-level optimization, based on measurable efforts and benefits. We argue that design-level optimization is necessary for achieving the long-term benefits, such as design and maintenance quality. However, they are often not explicitly measurable or immediately visible to practitioners. Therefore, more research is urgently needed to provide design guidance in resolving performance issues.

REFERENCES

[1] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE*

Transactions on Software Engineering, 30(5):295–310, 2004.

[2] Connie U Smith and Lloyd G Williams. Software performance antipatterns. In *Workshop on Software and Performance*, volume 17, pages 127–136. Ottawa, Canada, 2000.

[3] Connie U Smith and Lloyd G Williams. Software performance antipatterns; common performance problems and their solutions. In *Int. CMG Conference*, pages 797–806, 2001.

[4] Connie U Smith and Lloyd G Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, pages 717–725. Citeseer, 2003.

[5] Connie U Smith and Lloyd G Williams. New software performance antipatterns: More ways to shoot yourself in the foot. In *Int. CMG Conference*, pages 667–674, 2002.

[6] Guoqing Xu. Finding reusable data structures. In *ACM SIGPLAN Notices*, volume 47, pages 1017–1034. ACM, 2012.

[7] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *ACM Sigplan Notices*, volume 45, pages 174–186. ACM, 2010.

[8] Guoqing (Harry) Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008* [8], pages 151–160.

[9] Michael Pradel, Markus Huggler, and Thomas R Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 13–25. ACM, 2014.

[10] Luca Della Toffola, Michael Pradel, and Thomas R Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *ACM SIGPLAN Notices*, volume 50, pages 607–622. ACM, 2015.

[11] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 895–907. ACM, 2016.

[12] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 562–571. IEEE Press, 2013.

[13] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *ACM SIGPLAN Notices*, volume 50, pages 369–378. ACM, 2015.

[14] Lu Fang, Liang Dou, and Guoqing (Harry) Xu. Perfblower: Quickly detecting memory-related performance problems via amplification. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic* [14], pages 296–320.

[15] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009* [15], pages 397–407.

[16] Lixia Liu and Silvius Rus. Perflint: A context sensitive performance advisor for c++ programs. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 265–274. IEEE, 2009.

[17] Guoqing (Harry) Xu. Coco: Sound and adaptive replacement of java collections. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, pages 1–26, 2013.

[18] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 389–400. ACM, 2016.

[19] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in java. In *ACM SIGPLAN Notices*, volume 34, pages 35–46. ACM, 1999.

[20] Erik Ruf. Effective synchronization removal for java. In *ACM SIGPLAN Notices*, volume 35, pages 208–218. ACM, 2000.

[21] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 268–278. ACM, 2013.

[22] Ajeet Shankar, Matthew Arnold, and Rastislav Bodík. Jolt: lightweight dynamic analysis and removal of object churn. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA* [22], pages 127–142.

[23] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 902–912. IEEE, 2015.

[24] Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient flow profiling for detecting performance bugs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016* [24], pages 413–424.

[25] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. In *ACM Sigplan Notices*, volume 44, pages 408–418. ACM, 2009.

[26] Changhee Jung, Silvius Rus, Brian P Railing, Nathan Clark, and Santosh Pande. Brainy: effective selection of data structures. In *ACM SIGPLAN Notices*, volume 46, pages 86–97. ACM, 2011.

[27] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 237–246. IEEE Press, 2013.

[28] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, pages 61–72. ACM, 2016.

[29] Connie U Smith and Lloyd G Williams. Software performance antipatterns. In *Workshop on Software and Performance*, volume 17, pages 127–136. Ottawa, Canada, 2000.

[30] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012.

[31] https://pdfbox.apache.org/.

[32] https://avro.apache.org/.

[33] http://ant.apache.org/ivy/.

[34] https://commons.apache.org/proper/commons-collections/.

[35] http://groovy-lang.org/.

[36] https://issues.apache.org/jira/.

[37] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[38] Zhifei Chen, Bihuan Chen, Lu Xiao, Xiao Wang, Lin Chen, Yang Liu, and Baowen Xu. Speedoo: prioritizing performance optimization opportunities. In *Proceedings of the 40th International Conference on Software Engineering*, pages 811–821. ACM, 2018.

[39] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM, 2010.

[40] https://issues.apache.org/jira/browse/pdfbox-591.

[41] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 120–131. IEEE, 2016.

[42] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.

[43] Samireh Jalali and Claes Wohlin. Systematic literature studies: database searches vs. backward snowballing. In *Proceedings of the 2012 ACM-IEEE international symposium on empirical software engineering and measurement*, pages 29–38. IEEE, 2012.

[44] Carliss Young Baldwin and Kim B Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[45] Lu Xiao, Yuanfang Cai, and Rick Kazman. Design rule spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 967–977. ACM, 2014.

[46] Kevin J Sullivan, William G Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 99–108. ACM, 2001.

[47] https://issues.apache.org/jira/browse/pdfbox-1924.

[48] https://issues.apache.org/jira/browse/avro-753.

[49] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 270–281. ACM, 2015.

[50] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265. ACM, 2018.

[51] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. *ACM SIGPLAN Notices*, 47(6):89–98, 2012.

[52] Marc Brünink and David S Rosenblum. Mining performance specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 39–49. ACM, 2016.

[53] David Kawrykow and Martin P Robillard. Detecting inefficient api usage. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 183–186. IEEE, 2009.

[54] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. Api method recommendation without worrying about the task-api knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 293–304. ACM, 2018.

[55] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[56] David Kawrykow and Martin P Robillard. Improving api usage through automatic detection of redundant code. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 111–122. IEEE Computer Society, 2009.

[57] https://issues.apache.org/jira/browse/groovy-7977.

[58] https://issues.apache.org/jira/browse/avro-556.

[59] https://issues.apache.org/jira/browse/pdfbox-600.

[60] Diego Costa and Artur Andrzejak. Collectionswitch: A framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 16–26, 2018.

[61] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering*, pages 370–380. IEEE Press, 2017.

[62] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, and Michael A Kozuch. Rowclone: Fast and efficient in-dram copy and initialization of bulk data. 2013.

[63] Bruno Cartaxo, Paulo Borba, Sergio Soares, and Helio Fugimoto. Improving performance and maintainability of object cloning with lazy clones: An empirical evaluation. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–8. IEEE, 2015.

[64] Jungju Oh, Christopher J Hughes, Guru Venkataramani, and Milos Prvulovic. Lime: A framework for debugging load imbalance in multi-threaded execution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 201–210. ACM, 2011.

[65] Tongping Liu and Emery D Berger. Sheriff: precise detection and automatic mitigation of false sharing. *ACM Sigplan Notices*, 46(10):3–18, 2011.

[66] Tongping Liu, Chen Tian, Ziang Hu, and Emery D Berger. Predator: predictive false sharing detection. In *ACM SIGPLAN Notices*, volume 49, pages 3–14. ACM, 2014.

[67] Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 395–404. ACM, 2007.

[68] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, volume 5, pages 1–1, 2011.

[69] Bihuan Chen, Yang Liu, and Wei Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* [69], pages 49–60.

[70] https://issues.apache.org/jira/browse/pdfbox-893.

[71] https://issues.apache.org/jira/browse/pdfbox-3421.

[72] https://issues.apache.org/jira/browse/pdfbox-3224.

[73] https://issues.apache.org/jira/browse/pdfbox-604.

[74] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[75] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *ACM Sigplan Notices*, volume 45, pages 160–173. ACM, 2010.

[76] Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. Testing container classes: Random or systematic? In *International Conference on Fundamental Approaches to Software Engineering*, pages 262–277. Springer, 2011.

[77] Nicholas Hunt, Paramjit Singh Sandhu, and Luis Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *2011 15th Workshop on Interaction between Compilers and Computer Architectures*, pages 63–70. IEEE, 2011.

[78] Shengqian Yang, Dacong Yan, Guoqing Xu, and Atanas Rountev. Dynamic analysis of inefficiently-used containers. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, pages 30–35. ACM, 2012.

[79] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 389–400. ACM, 2017.

[80] Suparna Bhattacharya, Mangala Gowri Nanda, Kanchi Gopinath, and Manish Gupta. Reuse, recycle to de-bloat software. In *European Conference on Object-Oriented Programming*, pages 408–432. Springer, 2011.

[81] Guoqing Xu, Dacong Yan, and Atanas Rountev. Static detection of loop-invariant data structures. In *European Conference on Object-Oriented Programming*, pages 738–763. Springer, 2012.

[82] Alexander Wert, Jens Happe, and Lucia Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 552–561. IEEE Press, 2013.

[83] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 800–810. IEEE, 2018.

[84] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM, 2014.

[85] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *ACM SIGPLAN Notices*, volume 49, pages 561–578. ACM, 2014.

[86] Alejandro Infante. Identifying caching opportunities, effortlessly. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 730–732. ACM, 2014.

[87] Jian Li, Faheem Zafari, Don Towsley, Kin K Leung, and Ananthram Swami. Joint data compression and caching: Approaching optimality with guarantees. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 229–240. ACM, 2018.

[88] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 421–426. ACM, 2010.

[89] Thanh HD Nguyen, Meiyappan Nagappan, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. An industrial case study of automatically identifying performance regression-causes. In *Proceedings of the 11th working conference on mining software repositories*, pages 232–241, 2014.

[90] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering*, 42(12):1148–1161, 2016.

[91] Thomas Jensen, Florent Kirchner, and David Pichardie. Secure the clones. In *European Symposium on Programming*, pages 317–337. Springer, 2011.

[92] Yiqun Chen, Stefan Winter, and Neeraj Suri. Inferring performance bug patterns from developer commits.

[93] Yu Kang, Yangfan Zhou, Hui Xu, and Michael R Lyu. Persisdroid: Android performance diagnosis via anatomizing asynchronous executions. *arXiv preprint arXiv:1512.07950*, 2015.

[94] Akira Imai and Evan Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1030–1040, 1993.

[95] Achille Peternier, Walter Binder, Akira Yokokawa, and Lydia Chen. Parallelism profiling and wall-time prediction for multi-threaded applications. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 211–216. ACM, 2013.

[96] Sanath Jayasena, Saman Amarasinghe, Asanka Abeyweera, Gayashan Amarasinghe, Himeshi De Silva, Sunimal Rathnayake, Xiaoqiao Meng, and Yanbin Liu. Detection of false sharing using machine learning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 30. ACM, 2013.

[97] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGOPS Operating Systems Review*, 51(2):677–691, 2017.

[98] Stefano Conoci, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Adaptive performance optimization under power constraint in multi-thread applications with diverse scalability. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 16–27. ACM, 2018.