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[1,4]
Stevens Institute of Technology[1], Fudan University[2],
Nanyang Technological University[3],
Software Performance and Scalability Consulting LLC[4]

What is a Software Performance Issue?

- Software performance measures how effective is a software system with respect to *time constraints* and *allocation of resources*. [1]

- Performance issue happens when software fails to meet such requirements. Examples include:
  - Long time execution
  - Memory bloat
  - Program blocking

- “Users are more likely to switch to competitors’ products due to performance bugs than due to other general bugs.” [2]

Motivation

- Numerous prior studies investigated the **causes and solutions** of performance issues, with two limitations:
  - They usually only focused on a **specific type of problems.**
  - They mostly focus on performance issues that can be fixed by ***localized code changes***.

"Most performance issues have their roots in poor architectural decisions made before coding is done." [3]

---Smith & Williams

- We found that a significants (33%) portion of performance issues in the systems we examined require ***design-level*** optimization to ensure both performance improvement and code quality.

Research Questions

RQ 1: *What are the common root causes of real-life software performance issues? Is each type well-addressed in the existing literature?*

RQ 2: *Are performance issues addressed by design-level optimization? If so, how?*

RQ3: *What is the ROI (Return on Investment) for fixing performance issues?*

Key Contributions

- This study revealed 8 common root causes and resolutions to performance issues, and surveyed 60 related articles that investigated these root causes.
- This study provides empirical findings of design-level optimizations that are necessary for addressing performance issues.
- This study measures the Return on Investment for addressing performance issues.
- This study proposed a novel design structure modeling technique, named Diff Design Structure Matrix, for analyzing design-level optimizations.
- This study contributes a rich, high-quality dataset of 192 performance issues.

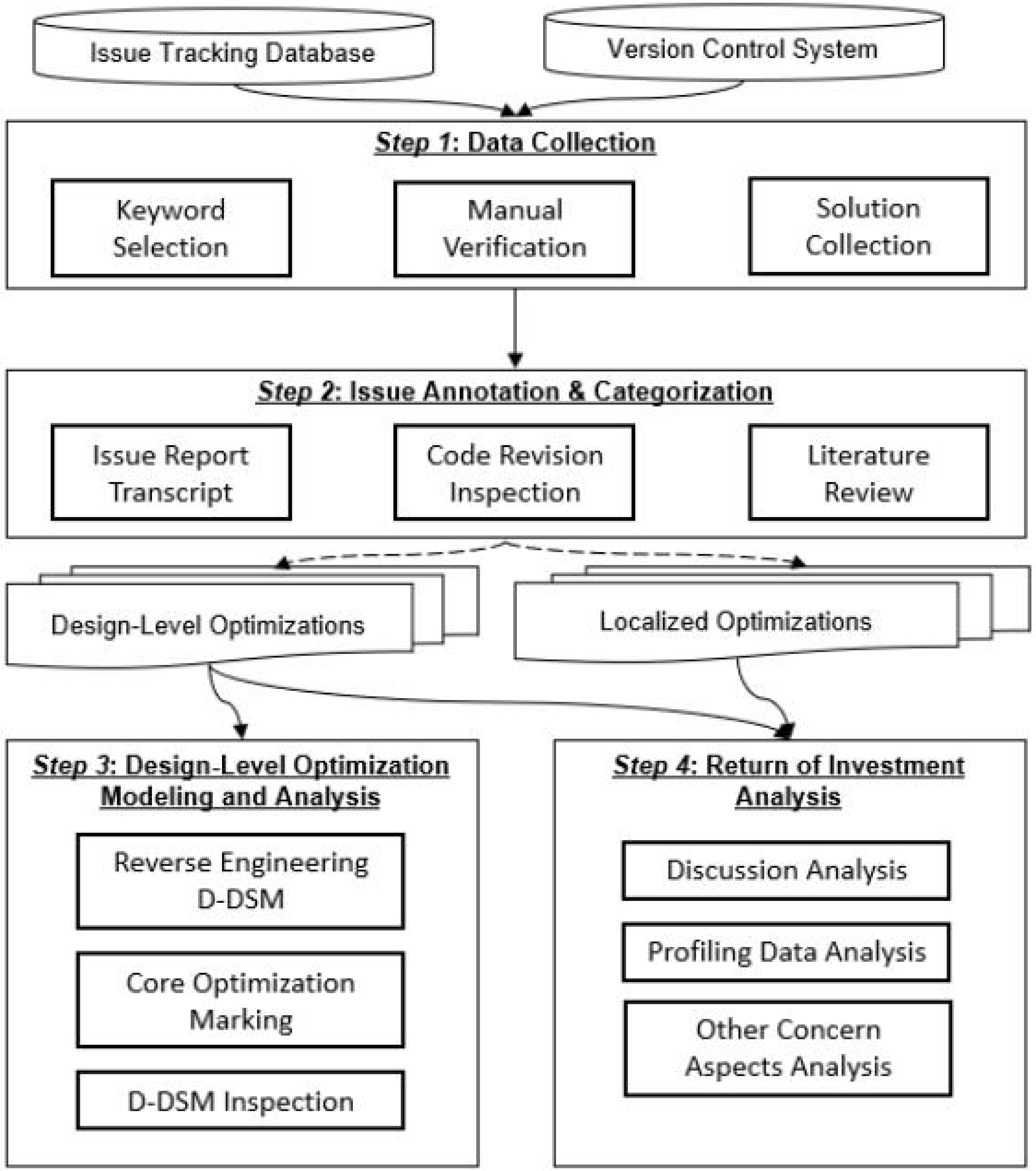
Study Projects

This study is based on five widely-used, open sourced projects from: THE APACHE SOFTWARE FOUNDATION

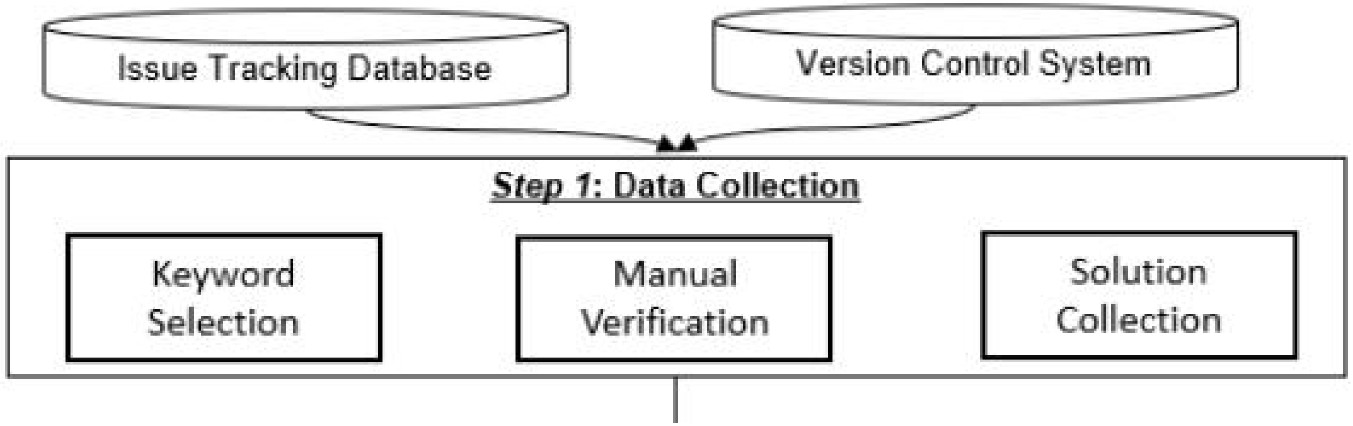
- **PDFBox**: Java tool working with PDF documents;
- **Avro**: remote data serialization framework;
- **Ivy**: transitive package manager to resolve complex project dependencies;
- **Collections**: Java collections library of Set, List, Map;
- **Groovy**: Java-syntax-compatible object-oriented programming language for Java platform.

Reasons: (1) In different domains;
(2) Performance is important;
(3) widely-used;
(4) code and discussion available.

Study Approach



Step 1: Data Collection



Issue Tracking System: PDFBox / PDFBOX-591

PDFBox performance issue: BaseParser.readUntilEndStream() rewrite

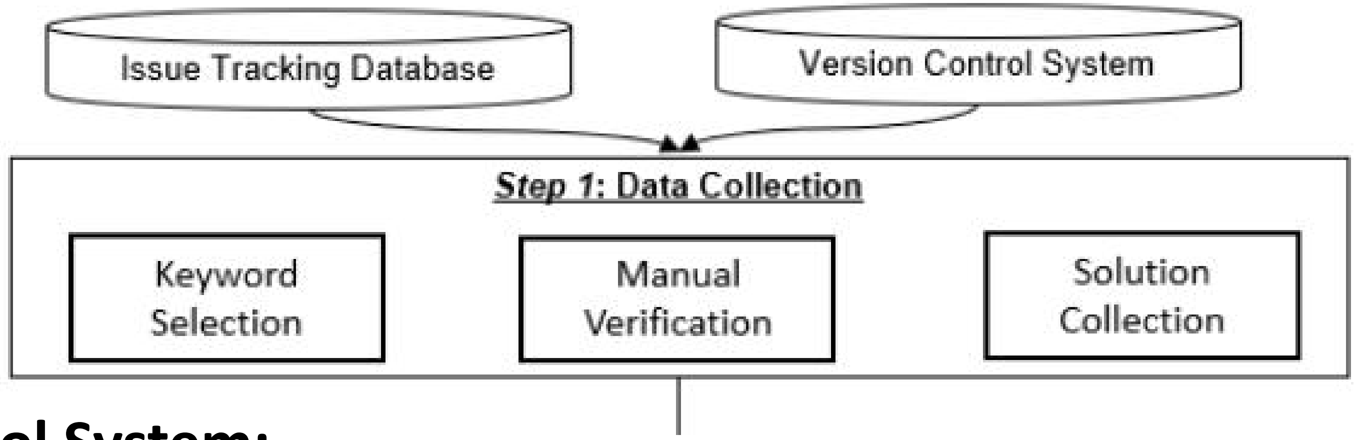
Description

The load time for loading documents into PDFBox (PDDocument) is too slow.

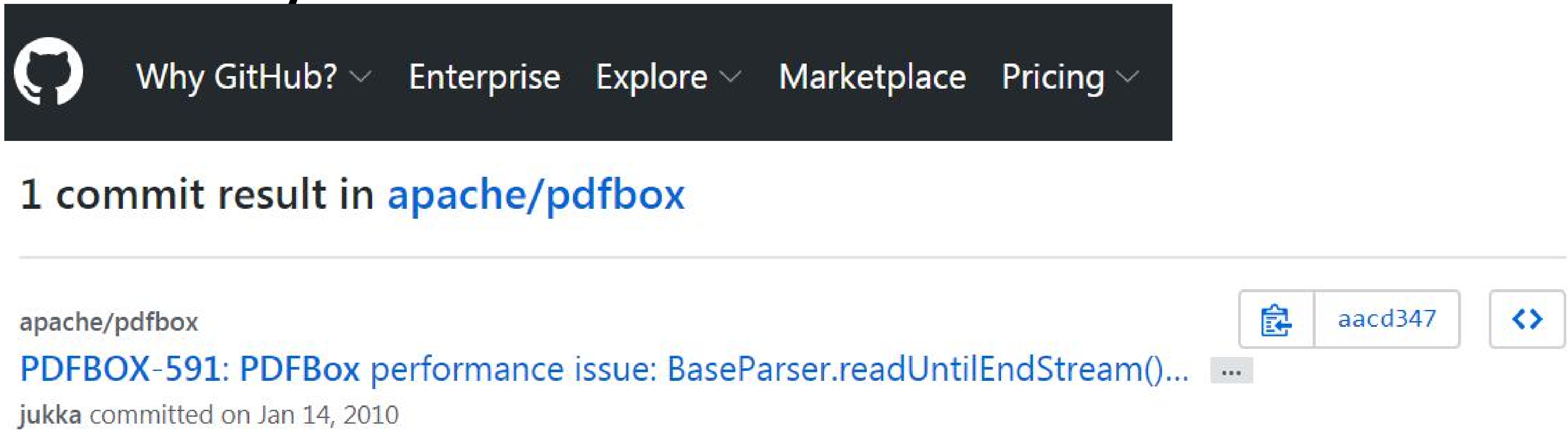
One culprit is the method: org.apach.pdfbox.pdfparser.BaseParser.readUntilEndStream(OutputStream out)

- **Keyword Selection**: *fast, slow, latency, speed, efficient, performance, unnecessary, redundant*, etc. (**512 selected**)
- **Manual Verification**: exclude false positives, e.g. “performance” can refer to productivity of developers. (**400 selected**)

Step 1: Data Collection

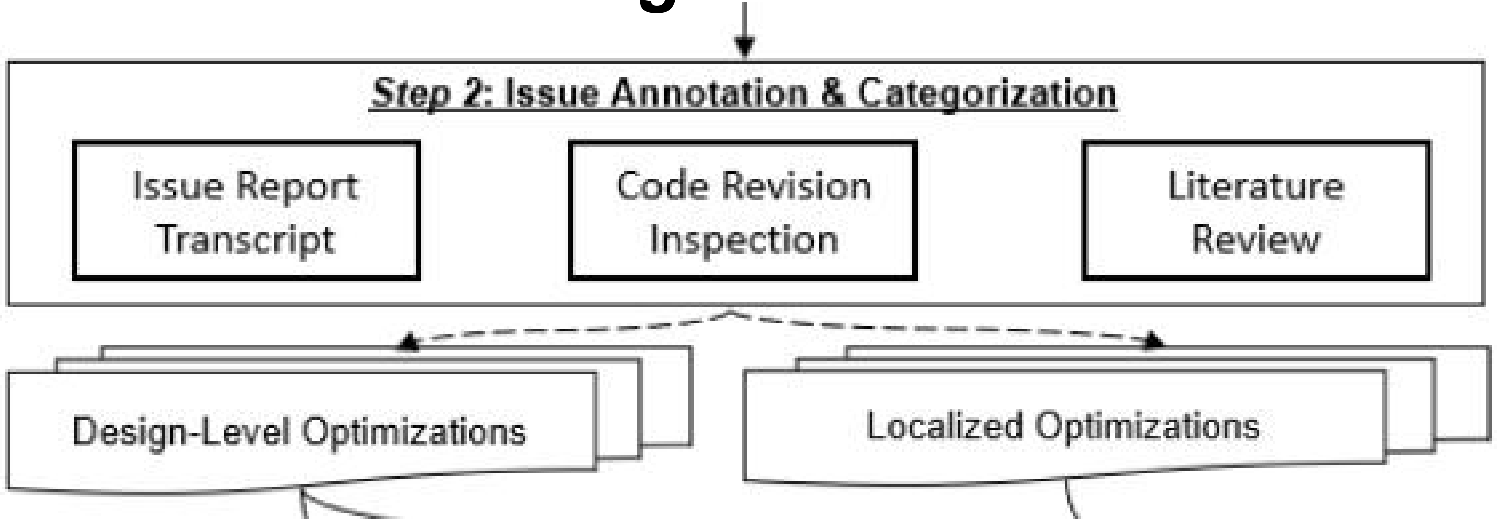


Version Control System:



- **Solution Collection**: extracted by issue ID. (**192 selected**)

Step 2: Issue Annotation & Categorization



- **Issue Report Transcript**: 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).
- **Code Revision Inspection**: reveal the most essential logic of the root causes and solutions to performance issues
- **Literature Review**: Keyword Search (Top 500) → Filtering (47) → Backward Snowballing (92)

  60 of them investigated root causes.

Localized Optimization

Localized Optimization: addressd by a few lines of code revision in a single source file.

```
protected void valid(COSDictionary action, boolean valid,
  String expectedCode) throws Exception {

  // process the action validation

  // check the result
  for (ValidationError err : errors) {
    if (err.getErrorCode().equals(expectedCode)) {
      found = true;
+         break;
    }
  }
  assertTrue(found);
}
```

PDFBOX-1459

Step 3: Design-Level Optimization Modeling and Analysis

Design-Level Optimization: a group of source files revised simultaneously for fixing performance-related reasons.

Calculation of D-DSM:

- Generate two versions of the code base (before and after the revision)
- Recover the structural dependencies among source files of the two versions
- Compare the dependencies and highlight the add/remove source files.

Diff Design Structural Matrix (D-DSM)

		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)

AVRO-753

Step 4: Return on Investment Analysis

- **Investment**: 1) Number of involved developers; 2) Number of Discussions

- **Return:**

$$1)\ \frac{ResponseTime\_BeforeFix}{ResponseTime\_AfterFix};$$

$$2)\ \frac{Throughput\_AfterFix}{Throughput\_BeforeFix}$$

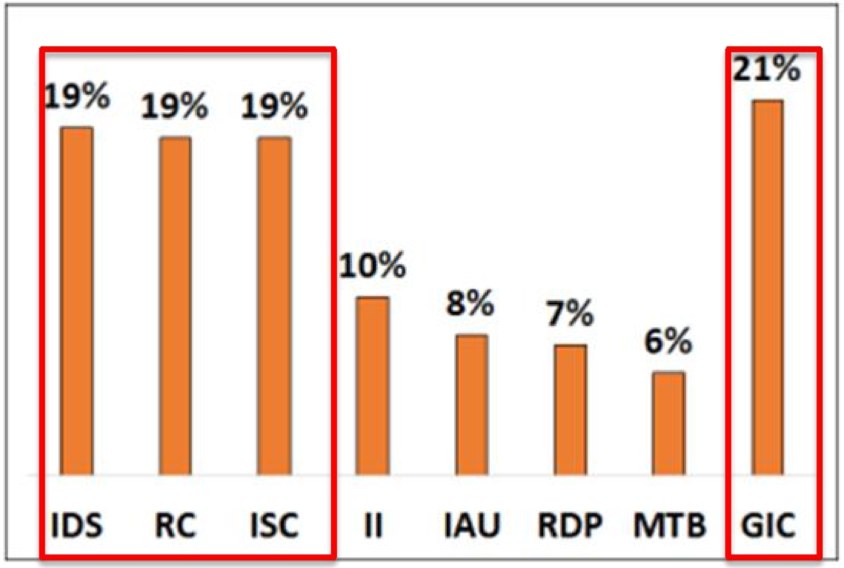
- We acknowledge that there are other meaningful measurements for investment and return.
- We focused on these metrics because they provide meaningful information and are easy to measure.

Study Result

RQ-1.1: What are the common root causes of performance issues?

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 after-thought.

IDS: Inefficient Data Structure
RC: Repeated Computation
ISC: Inefficiency under Special Cases
II: Inefficient Iteration
IAU: Inefficient API Usage
RDP: Redundant Data Processing
MTB: Multi-threaded Blocking
GIC: General Inefficient Computation



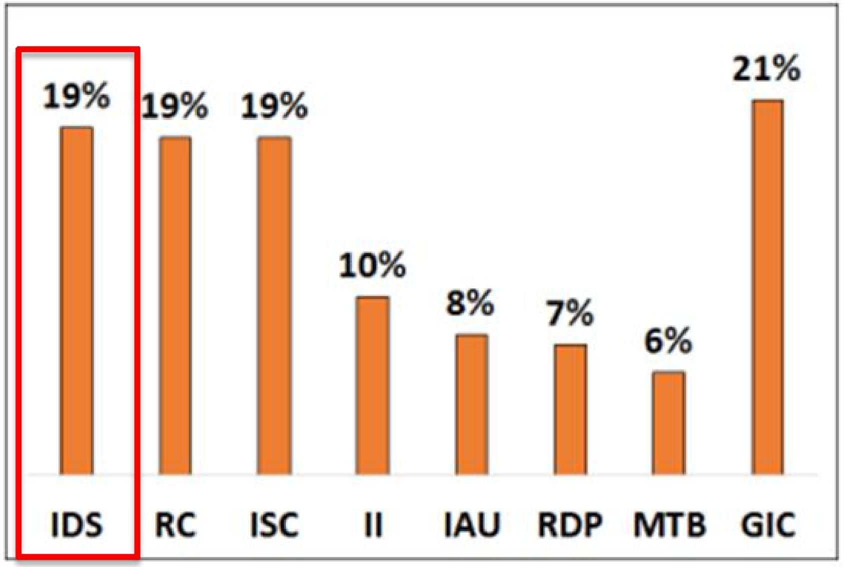
Prevalence of Different Root Causes

Study Result

RQ-1.1: What are the common root causes of performance issues?

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 after-thought.

IDS: Inefficient Data Structure
RC: Repeated Computation
ISC: Inefficiency under Special Cases
II: Inefficient Iteration
IAU: Inefficient API Usage
RDP: Redundant Data Processing
MTB: Multi-threaded Blocking
GIC: General Inefficient Computation



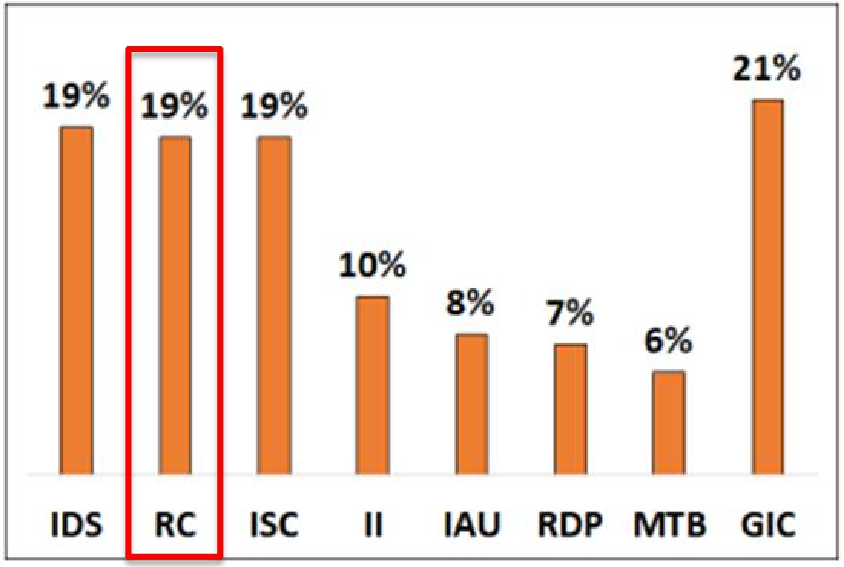
Prevalence of Different Root Causes

Study Result

RQ-1.1: What are the common root causes of performance issues?

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 after-thought.

IDS: Inefficient Data Structure
RC: Repeated Computation
ISC: Inefficiency under Special Cases
II: Inefficient Iteration
IAU: Inefficient API Usage
RDP: Redundant Data Processing
MTB: Multi-threaded Blocking
GIC: General Inefficient Computation



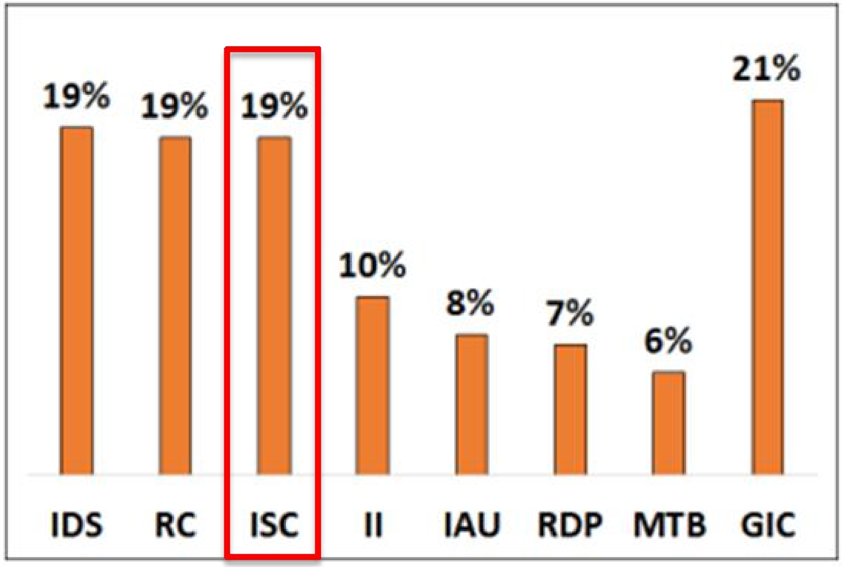
Prevalence of Different Root Causes

Study Result

RQ-1.1: What are the common root causes of performance issues?

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 after-thought.

IDS: Inefficient Data Structure
RC: Repeated Computation
ISC: Inefficiency under Special Cases
II: Inefficient Iteration
IAU: Inefficient API Usage
RDP: Redundant Data Processing
MTB: Multi-threaded Blocking
GIC: General Inefficient Computation



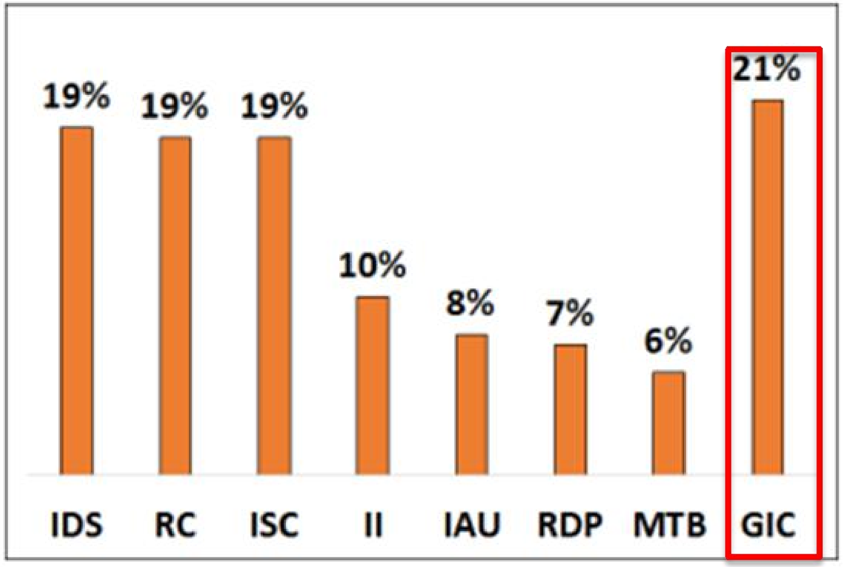
Prevalence of Different Root Causes

Study Result

RQ-1.1: What are the common root causes of performance issues?

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 after-thought.

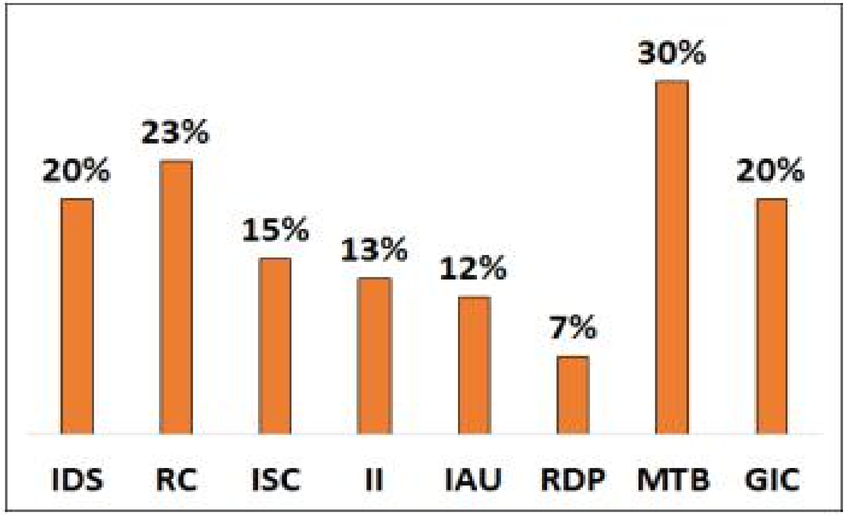
IDS: Inefficient Data Structure
RC: Repeated Computation
ISC: Inefficiency under Special Cases
II: Inefficient Iteration
IAU: Inefficient API Usage
RDP: Redundant Data Processing
MTB: Multi-threaded Blocking
GIC: General Inefficient Computation



Prevalence of Different Root Causes

Study Result

RQ-1.2: How well is each root cause addressed in the literature?



Prevalence in Literature

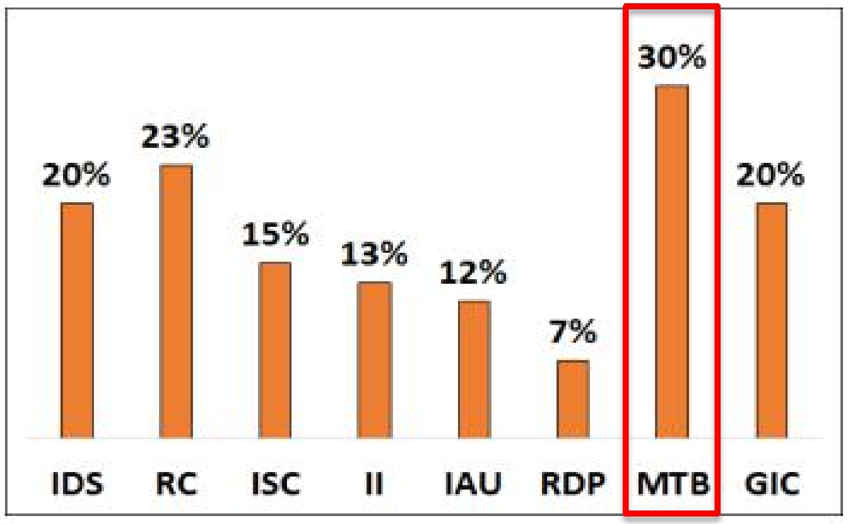
1) *Proposed tools have not been tested and compared to each other on large-scale, real-world dataset;*
2) *Tools are limited to Java/C/C++ projects;*
3) *The availability and usability of these tools are potential obstacles for practitioners to using them.*

Root Cause	Tool	Language	Year(A.)
Inefficient Data Structure	[D,F]:Perflint [16] [D,F] CoCo [17] [D,F]: CHAMELEON [25] [F]: Brainy [26]	C++ Java Java C++	2009(A) 2013 2009 2011
Repeated Computation	[D]: Cachetor [21] [D,F]: MemoizeIt [10]	Java Java	2013(A) 2015(A) 2015
Inefficiency under Special Cases	[D]: PerfFuzz [49] [D]: GA-Prof [48]	C Java	2018 2015
Inefficient Iteration	[D,F]: Caramel [23] [D]: Toddler [12] [D]: GLIDER [11] [D]: LDoctor [59] [F]: Clarity [13]	Java/C/C++ Java Java Java/C/C++ Java	2015 2013(A) 2016(A) 2017 2015
Inefficient API Usage	[D,F]: BIKER [53]	Java	2018
Redundant Data Processing	[F]: RowClone [60] [F]: LazyClone [61]	*assembly* Java	2013 2015
Multi-threaded Blocking	[D]: SpeedGun [9] [D,F]: SyncProf [18] [D]: LIME [62] [D,F]: SHERIFF [63] [D]: PRADATOR [64]	Java C/C++ C/C++ C/C++ C/C++	2014 2016 2011 2011(A) 2014(A)
General Inefficient Computation	[D]: Trend Profiler [65] [D]: Spectroscope [66] [D]: PerfPlotter [67]	C Perl/C++ Java	2007 2011(A) 2016(A)

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

Study Result

RQ-1.2: How well is each root cause addressed in the literature?



Prevalence in Literature

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
Repeated Computation	[D]: Cachetor [21]	Java	2013(A)
	[D,F]: MemoizeIt [10]	Java	2015(A)
			2015
Inefficiency under Special Cases	[D]: PerfFuzz [49]	C	2018
	[D]: GA-Prof [48]	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 [59]	Java/C/C++	2017
	[F]: Clarity [13]	Java	2015
Inefficient API Usage	[D,F]: BIKER [53]	Java	2018
Redundant Data Processing	[F]: RowClone [60]	*assembly*	2013
	[F]: LazyClone [61]	Java	2015
Multi-threaded Blocking	[D]: SpeedGun [9]	Java	2014
	[D,F]: SyncProf [18]	C/C++	2016
	[D]: LIME [62]	C/C++	2011
	[D,F]: SHERIFF [63]	C/C++	2011(A)
	[D]: PRADATOR [64]	C/C++	2014(A)
General Inefficient Computation	[D]: Trend Profiler [65]	C	2007
	[D]: Spectroscope [66]	Perl/C++	2011(A)
	[D]: PerfPlotter [67]	Java	2016(A)

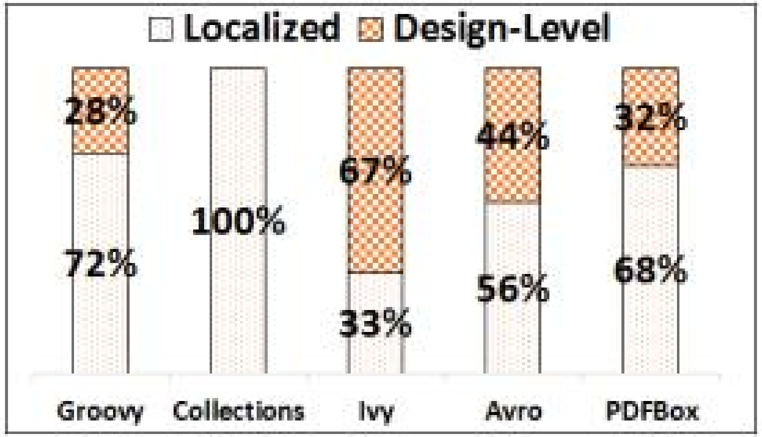
Note: "D" means the tool can automatically **detect**.
"F" means the tool can automatically provide **fixing** resolutions.

1) *Proposed tools have not been tested and compared to each other on large-scale, real-world dataset;*
2) *Tools are limited to Java/C/C++ projects;*
3) *The availability and usability of these tools are potential obstacles for practitioners to using them.*

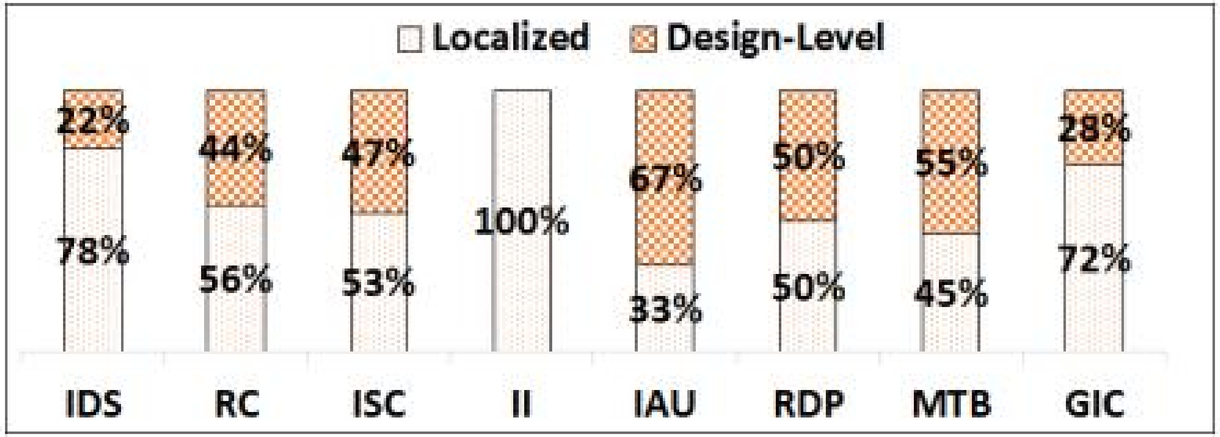
Study Result

RQ-2.1: Are performance issues usually addressed by localized optimization or complicated design-level optimization?

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.



(a) By Project



(b) By Root Cause

Study Result

RQ-2.2: What are the typical design-level optimization patterns?

- **Classic Design Patterns**: The developers employ classical design patterns for addressing the performance issues and achieving good design at the same time.

		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

Study Result

RQ-2.2: What are the typical design-level optimization patterns?

- **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.

		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

Study Result

RQ-2.2: What are the typical design-level optimization patterns?

- **Optimization Clone**: The developers fix multiple instances of the same performance root cause that are cloned in multiple locations in the code base.

Inefficient method, *getBoundingBox()*, is cloned in these seven files.

		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

Answer to RQ-2

RQ-2.2: What are the typical design-level optimization patterns?

- **Parallel Optimization:** The developers made parallel optimizations in multiple locations that suffer from different root causes for resolving an issue.

1) **PDFont**: add cache to memorize font type to avoid repeated computation.
2) **PDSimpleFont**: avoid duplicate *has()* lookups.
3) **COSNumber**: Use a direct table lookup instead of a hash map to speed up COSNumber.get().
4) **ICU4HImpl**: only allocate a new buffer when one really is needed.
5) **PDFStreamEngine**: Use StringBuilder and Arrays.fill() instead of StringBuffer and an explicit loop to speed up

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

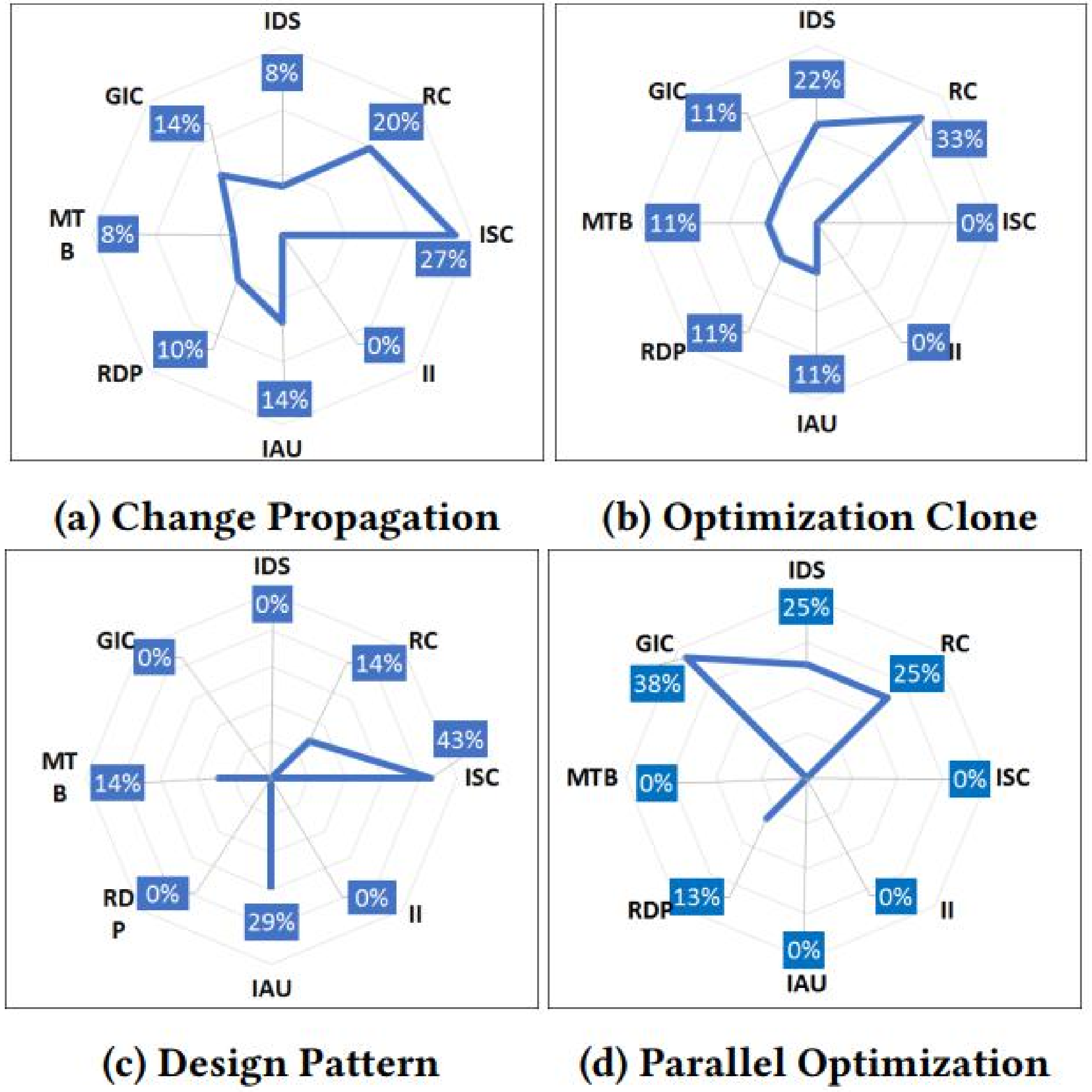
(e) Parallel Optimization: PDFBox-604

Answer to RQ-2

RQ-2.3: How prevalent is each design-level optimization pattern, especially for addressing different root causes?

- The applications of the four patterns for addressing different from each other.

- Inefficient iterations are excluded in this discussion, because they are only addressed by localized optimization.



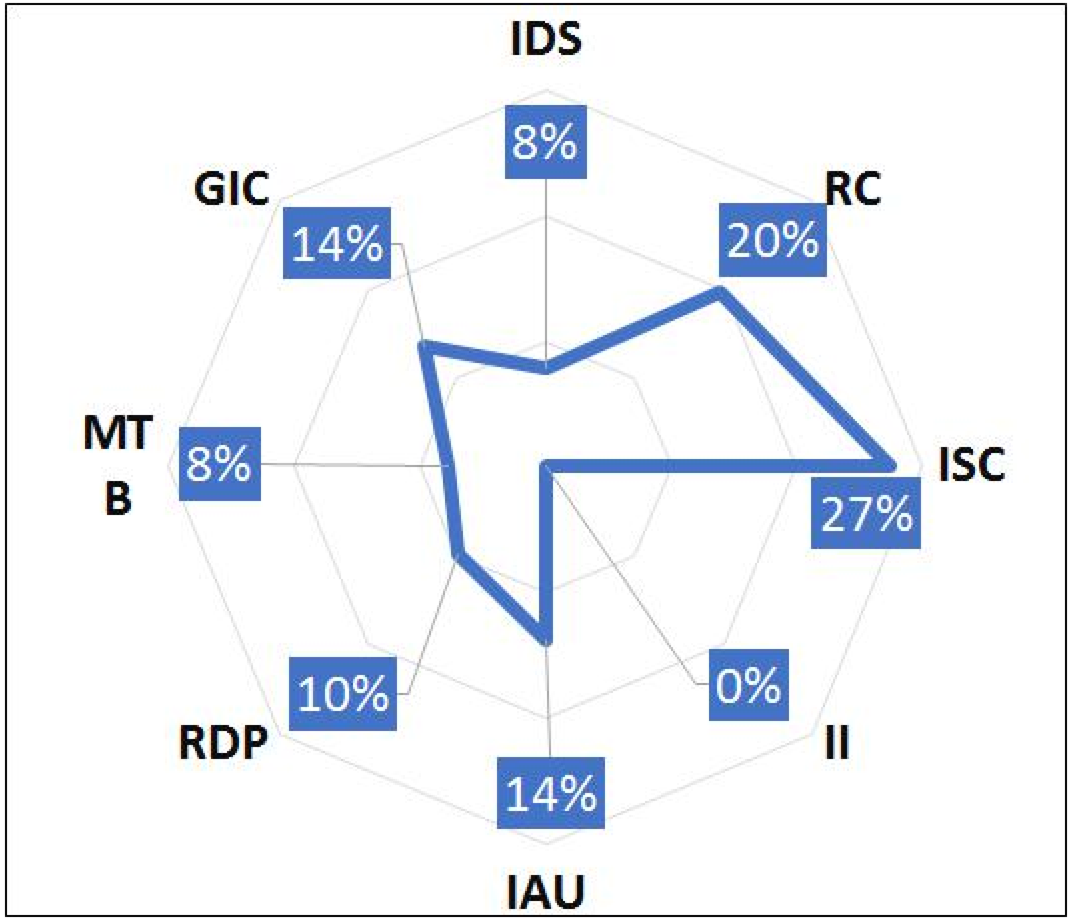
(a) Change Propagation (b) Optimization Clone

(c) Design Pattern (d) Parallel Optimization

Answer to RQ-2

- The majority (41% in Type-I, 27% in Type-II) of design-level optimizations are change propagations.

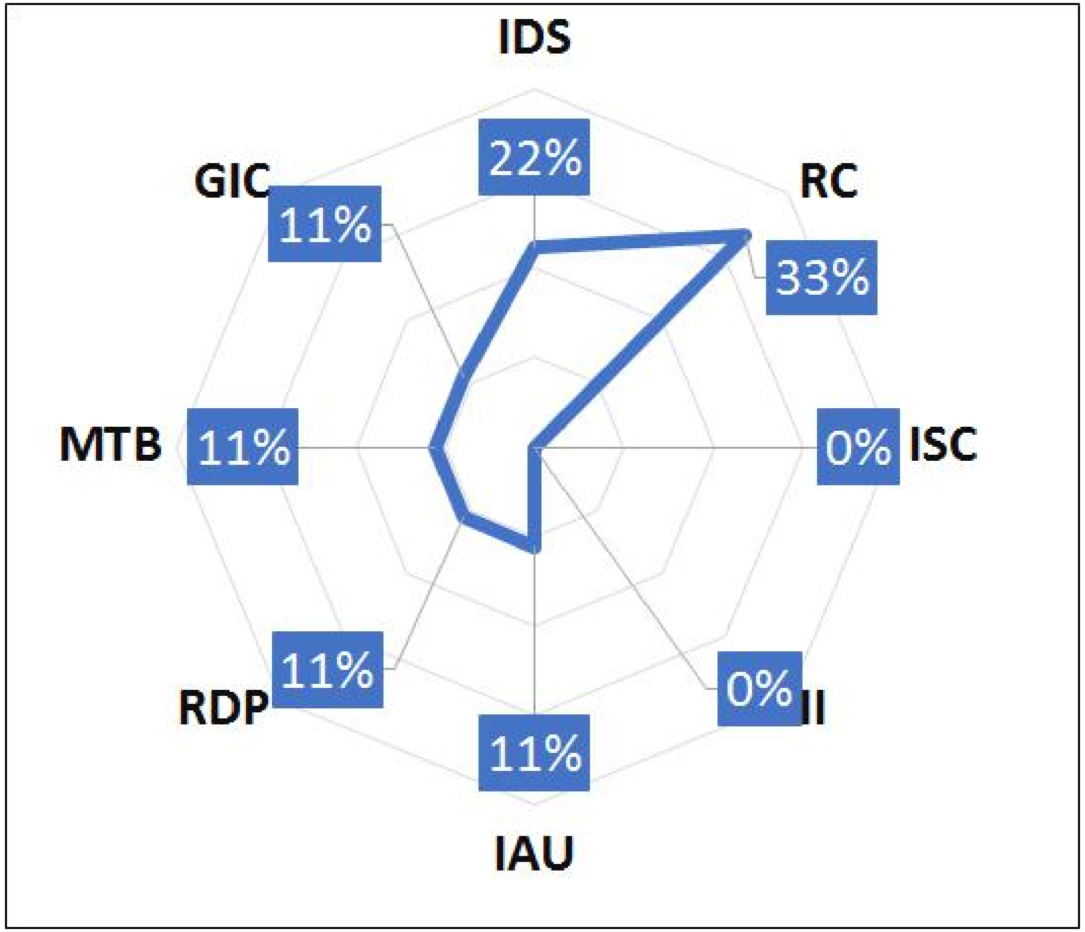
- All different types of root causes can be applied to address it.



(a) Change Propagation

Answer to RQ-2

- Optimization clone is not applied for addressing inefficiency under special cases (ISC).
- We conjecture that it is because special cases should be treated specifically so that the optimization would not be cloned.

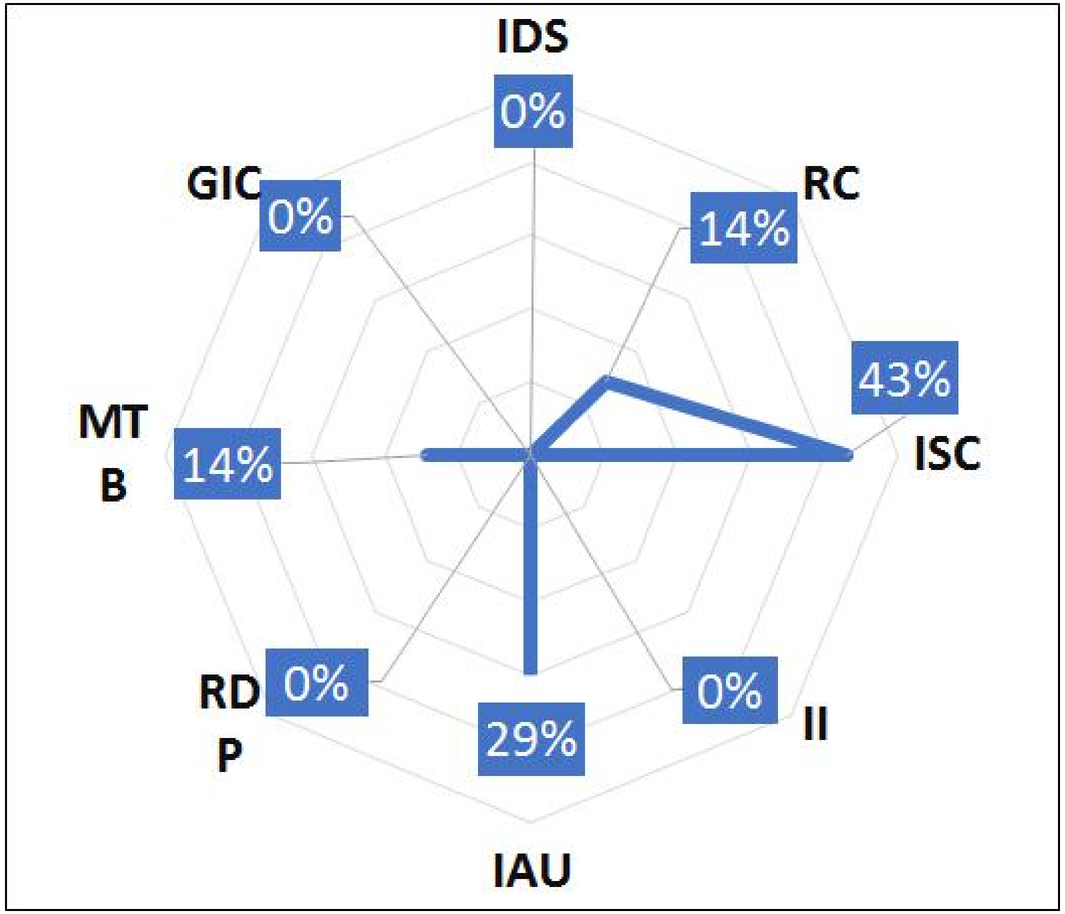


(b) Optimization Clone

Answer to RQ-2

- Classic design patterns are not applied for addressing inefficient data structure (IDS) and general inefficient computation (GIC).

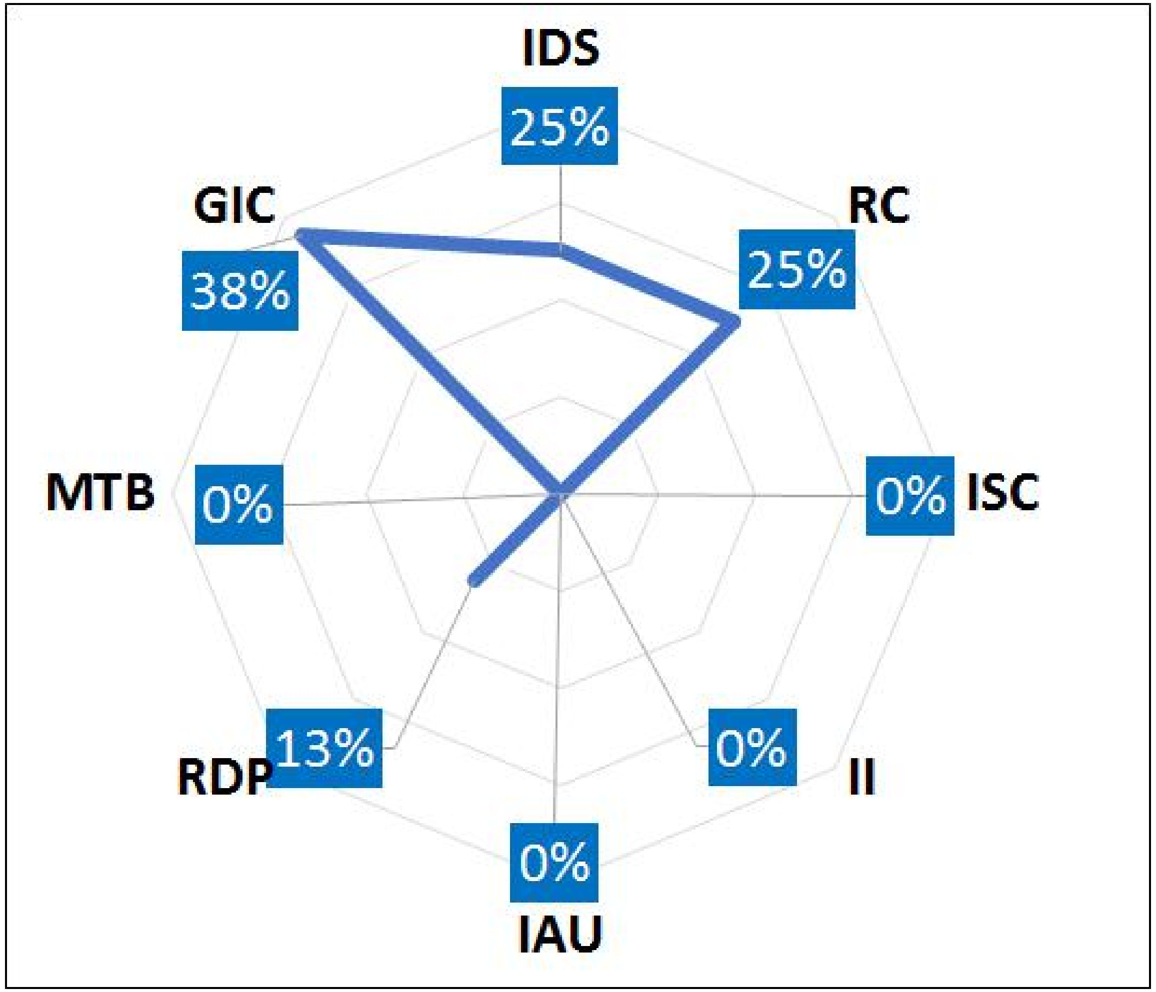
- We conjecture that it is because data structure and algorithmic optimization are usually located inside a single source file.



(c) Classic Design Pattern

Answer to RQ-2

- Parallel optimization mainly applies for general inefficient computation (GIC), inefficient data structure (IDS), and repeated computation (RC).
- We conjecture it is because these three root causes can be resolved by short code revisions.

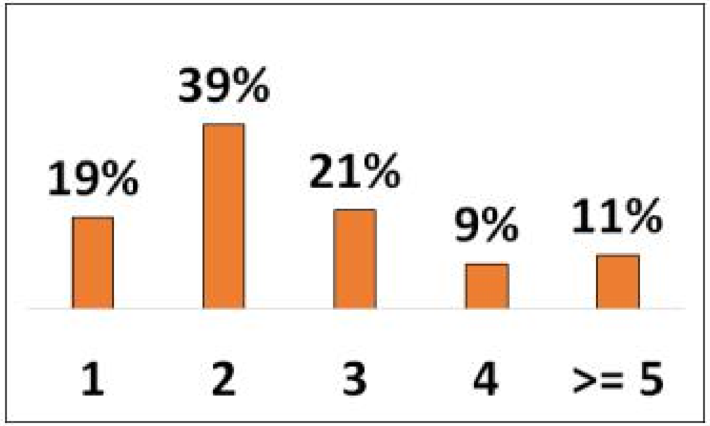


(d) Parallel Optimization

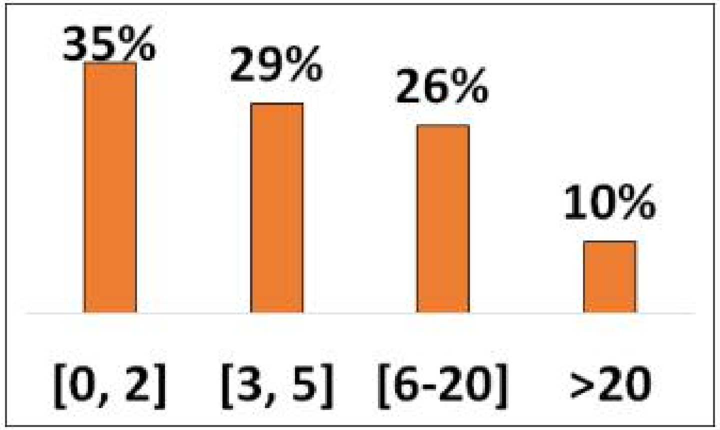
Answer to RQ-3

RQ-3.1 What is the overall ROI for addressing performance issues?

- **Investment**: 1) Number of involved developers; 2) Number of Discussions



(a) # Developers

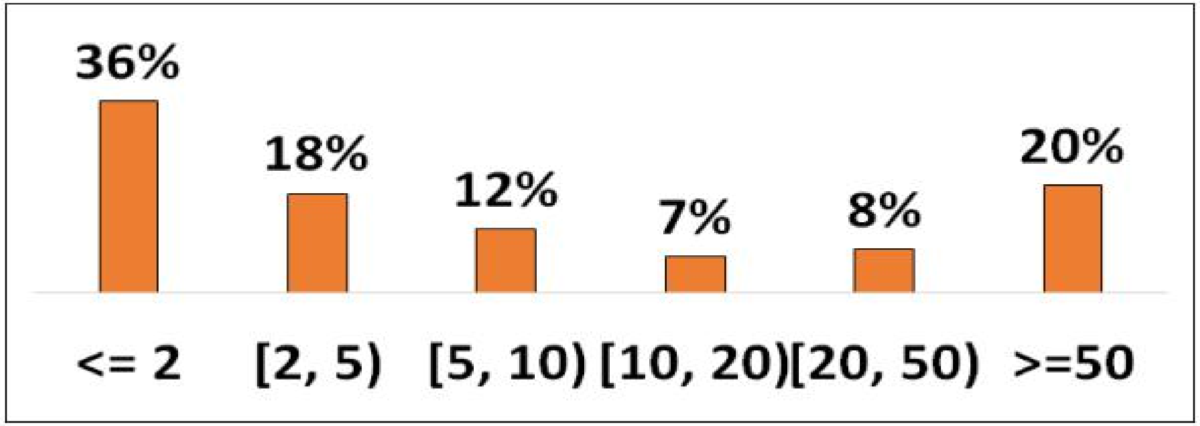


(b) # Discussions

- **Improvement:**

1) $\frac{ResponseTime\_BeforeFix}{ResponseTime\_AfterFix}$

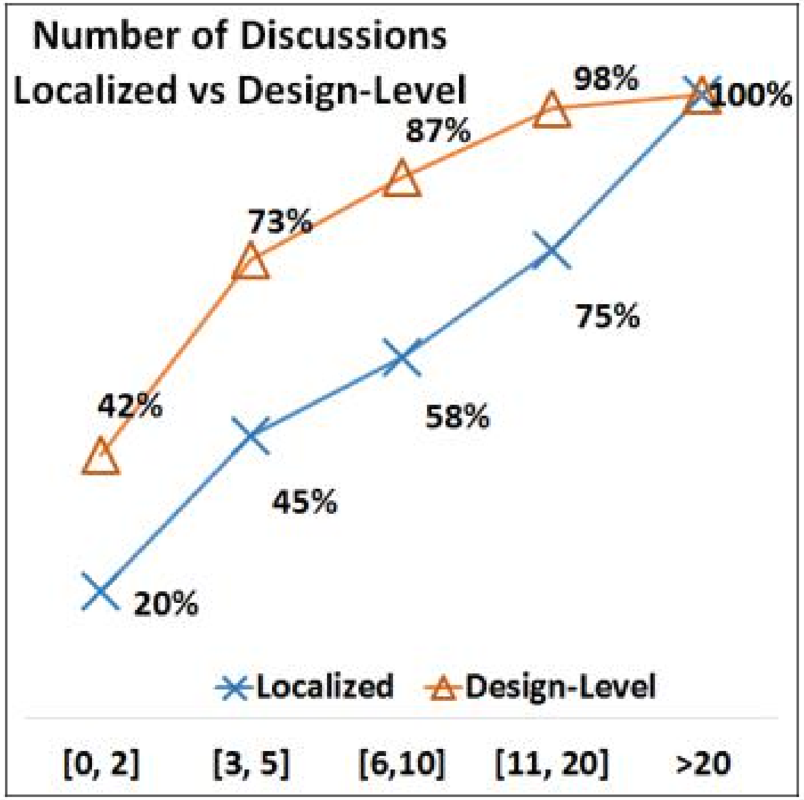
2) $\frac{Throughput\_AfterFix}{Throughput\_BeforeFix}$

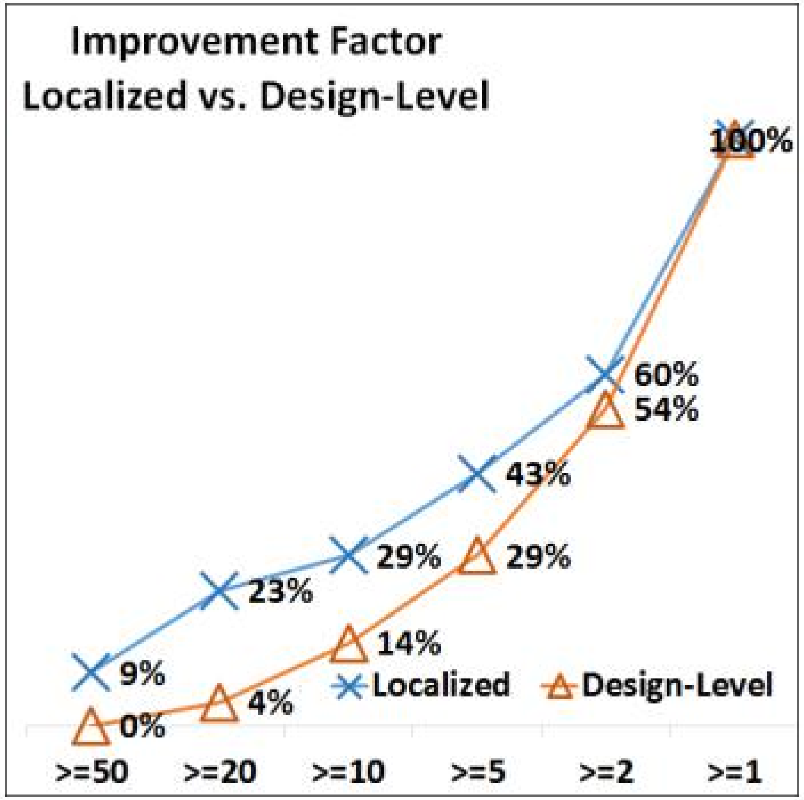


(c) Improvement

Answer to RQ-3

RQ-3.2 How is the ROI of localized and design-level optimization compared to each other?



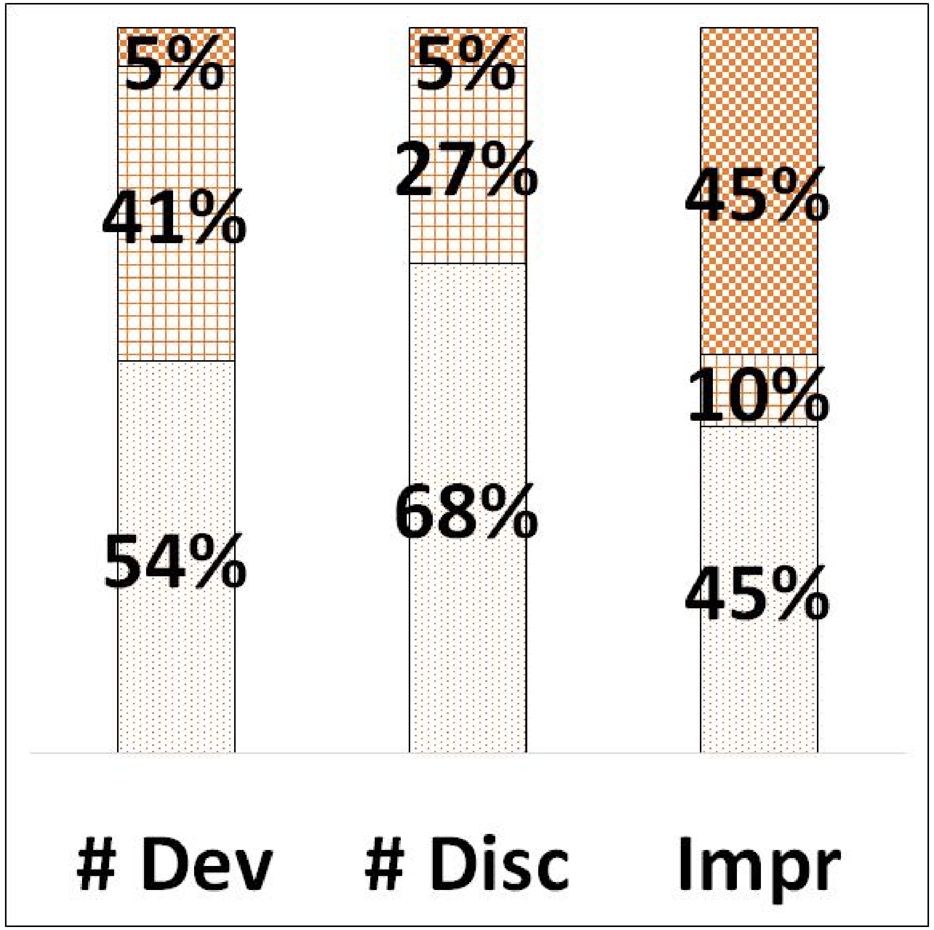


(a) Investment **(b) Improvement**

We conjecture that design-level optimization will provide benefits other than performance improvement, e.g. readability and maintainability—73% of these issues employed design-level optimization.

Answer to RQ-3

RQ-3.3 How is the ROI of performance issues affected by different root causes?



ROI of Inefficient Data Structure

Legend	# Developers	# Discussions	Improvement Factor
(light dotted)	[1, 2]	[0, 5]	[1x, 10x]
(grid)	[3, 4]	[6, 20]	[11x, 50x]
(dense checkered)	>= 5	> 20	> 50x

Legend

Limitations & Future Work

Limitations:

- We did not evaluate the actual effectiveness and usability of the fixing and detecting tools.
- The performance improvement is evaluated based on the available profiling data contained in the issue reports.
- We acknowledge that there are other meaningful measurements for Return on Investment.

Future Work:

- We plan to collect and use the detecting and fixing tools in prior studies in our dataset.
- We will try to evaluate the improvement of all the 192 performance issues by executing the code.
- We will investigate the impact of programming language on performance issues and their Return on Investment.

Conclusion

- This study investigate 192 real-life performance issues, and identified eight recurring root causes and typical resolutions.
- 33% of investigated performance issues require design-level optimization, manifested in four different typical patterns.
- Localized optimizations provide higher Return on Investment than design-level optimizations, based on measurable efforts and benefits.
- We argue that design-level optimization is necessary for achieving long-term benefits, such as good design and maintenance quality.

References

[1] Cortellessa, V., & Frittella, L. (2007, September). A framework for automated generation of architectural feedback from software performance analysis. In European Performance Engineering Workshop (pp. 171-185). Springer, Berlin, Heidelberg.

[2] Zaman, Shahed, Bram Adams, and Ahmed E. Hassan. "A qualitative study on performance bugs." *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 2012.

[3] Connie U Smith and Lloyd G Williams. Software performance anti-patterns. In Workshop on Software and Performance, volume 17, pages 127–136. Ottawa, Canada, 2000.

[4] .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.

[5] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pages 416–419. ACM, 2011.

[6] 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.

[7] Zhao, Y., Xiao, L., Xiao, W., Chen, B., & Liu, Y. (2019, May). Localized or architectural: an empirical study of performance issues dichotomy. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) (pp. 316-317). IEEE.


THANK YOU!
ANY QUESTIONS?