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

Yutong Zhao\textsuperscript{1}, Lu Xiao\textsuperscript{1}, Xiao Wang\textsuperscript{1}, Lei Sun\textsuperscript{1}, Bihuan Chen\textsuperscript{2}, Yang Liu\textsuperscript{3}, Andre B. Bondi\textsuperscript{1,4}

\textsuperscript{1}Stevens Institute of Technology, \textsuperscript{2}Fudan University, \textsuperscript{3}Nanyang Technological University, \textsuperscript{4}Software Performance and Scalability Consulting LLC
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 significant portion (33%) 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:

- **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**
- Keyword Selection
- Manual Verification
- Solution Collection

**Step 2: Issue Annotation & Categorization**
- Issue Report Transcript
- Code Revision Inspection
- Literature Review

**Step 3: Design-Level Optimization**
- Modeling and Analysis
  - Reverse Engineering D-DSM
  - Core Optimization Marking
  - D-DSM Inspection

**Step 4: Return of Investment Analysis**
- Discussion Analysis
- Profiling Data Analysis
- Other Concern Aspects Analysis
Step 1: Data Collection

Issue Tracking System: PDFBox performance issue: BaseParser.readUntilEndStream() rewrite

- **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:

1 commit result in apache/pdfbox

- **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: addressed by a few lines of code revision in a single source file.

```java
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);
}
```
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.
Step 4: Return on Investment Analysis

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

- **Return**:  
  \[
  \frac{\text{ResponseTime\_BeforeFix}}{\text{ResponseTime\_AfterFix}} ;
  \]
  \[
  \frac{\text{Throughput\_AfterFix}}{\text{Throughput\_BeforeFix}}
  \]

- 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
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.

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

- **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](image-url)
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.

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

- **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](chart.png)
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
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.

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

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?**

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.

### Prevalence in Literature

<table>
<thead>
<tr>
<th>Root Cause</th>
<th>Tool</th>
<th>Language</th>
<th>Year(A.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inefficient Data Structure</td>
<td>[D,F]:Perfint [16]</td>
<td>C++</td>
<td>2009(A)</td>
</tr>
<tr>
<td></td>
<td>[F]: Brainy [26]</td>
<td>C++</td>
<td>2011</td>
</tr>
<tr>
<td>Repeated Computation</td>
<td>[D]: Cachetor [21]</td>
<td>Java</td>
<td>2013(A)</td>
</tr>
<tr>
<td>Inefficiency under</td>
<td>[D]: PerfFuzz [49]</td>
<td>C</td>
<td>2018</td>
</tr>
<tr>
<td>Special Cases</td>
<td>[D]: GA-Prof [48]</td>
<td>Java</td>
<td>2015</td>
</tr>
<tr>
<td></td>
<td>[D]: Toddler [12]</td>
<td>Java</td>
<td>2013(A)</td>
</tr>
<tr>
<td></td>
<td>[D]: LDoctor [59]</td>
<td>Java/C/C++</td>
<td>2017</td>
</tr>
<tr>
<td>Inefficient API Usage</td>
<td>[D,F]: BIKER [53]</td>
<td>Java</td>
<td>2018</td>
</tr>
<tr>
<td>Redundant Data Processing</td>
<td>[F]: RowClone [60]</td>
<td>assembly</td>
<td>2013</td>
</tr>
<tr>
<td></td>
<td>[F]: LazyClone [61]</td>
<td>Java</td>
<td>2015</td>
</tr>
<tr>
<td></td>
<td>[D,F]: SHERIFF [63]</td>
<td>C/C++</td>
<td>2011(A)</td>
</tr>
<tr>
<td></td>
<td>[D]: PRADATOR [64]</td>
<td>C/C++</td>
<td>2014(A)</td>
</tr>
<tr>
<td>General Inefficient</td>
<td>[D]: Trend Profiler [65]</td>
<td>C</td>
<td>2007</td>
</tr>
<tr>
<td>Computation</td>
<td>[D]: Spectroscop [66]</td>
<td>Perl/C++</td>
<td>2011(A)</td>
</tr>
<tr>
<td></td>
<td>[D]: PerfPlotter [67]</td>
<td>Java</td>
<td>2016(A)</td>
</tr>
</tbody>
</table>

Note: "D" means the tool can automatically detect.
"F" means the tool can automatically provide fixing resolutions.
RQ-1.2: How well is each root cause addressed in the 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.
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.
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.

(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.
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.

(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

(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.
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.
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.
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.
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

- **Improvement:**
  1) \( \frac{\text{ResponseTime}_{\text{Before Fix}}}{\text{ResponseTime}_{\text{After Fix}}} \)
  2) \( \frac{\text{Throughput}_{\text{After Fix}}}{\text{Throughput}_{\text{Before Fix}}} \)
Answer to RQ-3

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

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

<table>
<thead>
<tr>
<th>Legend</th>
<th># Developers</th>
<th># Discussions</th>
<th>Improvement Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Legend 1]</td>
<td>[1, 2]</td>
<td>[0, 5]</td>
<td>[1x, 10x]</td>
</tr>
<tr>
<td>![Legend 2]</td>
<td>[3, 4]</td>
<td>[6, 20]</td>
<td>[11x, 50x]</td>
</tr>
<tr>
<td>![Legend 3]</td>
<td>&gt;= 5</td>
<td>&gt; 20</td>
<td>&gt; 50x</td>
</tr>
</tbody>
</table>
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


THANK YOU!

ANY QUESTIONS?