UNDERSTANDING, DETECTING, AND REPAIRING PERFORMANCE BUGS

BY

ADRIAN NISTOR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Associate Professor Darko Marinov, Chair
Assistant Professor Shan Lu, University of Wisconsin-Madison
Professor Josep Torrellas
Associate Professor Tao Xie

Abstract

Software performance is critical for how end-users perceive the quality of software products. Performance bugs—programming errors that cause performance degradation—lead to poor user experience and low system throughput. Despite advances in profiling techniques, performance bugs still escape in production runs. There are two key reasons why performance bugs are not effectively detected during in-house testing. First, there is little available data about how performance bugs are discovered, reported, and fixed in practice. Such data is required when designing effective techniques for addressing performance bugs. Second, the current techniques for detecting performance bugs detect only slow computation and do not address other important parts of the testing process, such as automated oracles or bug fixing.

This dissertation makes three contributions. The first contribution is a study of how performance bugs are discovered, reported to developers, and fixed by developers, and how these results compare with the results for non-performance bugs. The study considers performance and non-performance bugs from three popular code bases: Eclipse JDT, Eclipse SWT, and Mozilla. First, we find little evidence that fixing performance bugs has a higher chance to introduce new functional bugs than fixing non-performance bugs, which implies that developers may not need to be over-concerned about fixing performance bugs. Second, although fixing performance bugs is about as error-prone as fixing non-performance bugs, fixing performance bugs is more difficult than fixing non-performance bugs, indicating that developers need better tool support for fixing performance bugs and testing performance bug patches. Third, unlike many non-performance bugs, a large percentage of performance bugs are discovered through code reasoning, not through users observing the negative effects of the bugs (e.g., performance degradation) or through profiling. The result

suggests that techniques to help developers reason about performance, better test oracles, and better profiling techniques are needed for discovering performance bugs.

The second contribution is TODDLER, a novel automated oracle for performance bugs, which enables testing for performance bugs to use the well established and automated process of testing for functional bugs. TODDLER reports code loops whose computation has repetitive and partially similar memory-access patterns across loop iterations. Such repetitive work is likely unnecessary and can be done faster. TODDLER was implemented for Java and evaluated on 9 popular Java codebases. The experiments with 11 previously known, real-world performance bugs show that TODDLER finds these bugs with a higher accuracy than the standard Java profiler. TODDLER also found *42 new bugs* in six Java projects: Ant, Google Core Libraries, JUnit, Apache Collections, JDK, and JFreeChart. Based on the corresponding bug reports, developers so far fixed 10 bugs and confirmed 6 more as real bugs.

The third contribution is LULLABY, a novel static technique that detects and fixes performance bugs that have non-intrusive fixes likely to be adopted by developers. Each performance bug detected by LULLABY is associated with a loop and a condition. When the condition becomes true during the loop execution, all the remaining computation performed by the loop is wasted. Developers typically fix such performance bugs because these bugs waste computation in loops and have non-intrusive fixes: when some condition becomes true dynamically, just break out of the loop. Given a program, LULLABY detects such bugs statically and gives developers a potential source-level fix for each bug. LULLABY was evaluated on *real-world applications*, including 11 popular Java applications (e.g., Groovy, Log4J, Lucene, Struts, Tomcat, etc) and 4 widely used C/C++ applications (Chromium, GCC, Mozilla, and MySQL). LULLABY finds *61 new performance bugs* in the Java applications and *89 new performance bugs* in the C/C++ applications. Based on the corresponding bug reports, *developers so far have fixed 51 and 65 performance bugs* in the Java and C/C++ applications, respectively. Most of the remaining bugs are still under consideration by developers.

Acknowledgments

I could not have written this dissertation without my co-authors and my collaborators. Prof. Darko Marinov, Prof. Josep Torrellas, and Prof. Shan Lu taught me how to do research: discuss ideas, find problems, propose solutions, and write papers. In addition, I thank:

- Prof. Darko Marinov for great advice and continuous support.
- Prof. Josep Torrellas for introducing me to top-quality research.
- Prof. Shan Lu for being my model of a researcher.
- Prof. Lin Tan for a wonderful collaboration.
- Prof. Miryung Kim for graciously sharing her data.
- Prof. Vikram Adve and Prof. Tao Xie for their invaluable feedback.
- Qingzhou Luo, Michael Pradel, Linhai Song, Tian Jiang, Po-Chun Chang, Cosmin Radoi, and Lenin Sivalingam for making my projects better.
- Jihun Park for graciously sharing his data.
- Shanxiang Qi, Xuehai Qian, Ulya Karpuzcu, Wonsun Ahn, Brian Greskamp, Steven Lauterburg, Vilas Jagannath, Samira Tasharofi, Stas Negara, Nick Chen, Rakesh Komuravelli, Hyojin Sung, and Byn Choi for help and useful discussions.

Many other people contributed and should be acknowledged here. I appreciate their help.

Table of Contents

Chapter 1

Introduction

Software performance is important to the overall success of a software project. Performance bugs—programming errors that create significant performance degradation [12]—hurt software performance and quality[1]. Performance bugs lead to poor user experience, degrade application responsiveness, lower system throughput, and waste computational resources [11, 55]. Even expert programmers introduce performance bugs, which have already caused serious problems [39,54,56,70]. Well tested commercial products such as Internet Explorer, Microsoft SQLServer, and Visual Studio are also affected by performance bugs [52].

While there has been a lot of research on performance profiling and there are many mature commercial and open-source profiling tools—tools that identify slow code—available [1, 5, 53, 67, 90], performance bugs still escape into production runs. There are two key reasons for this current state of practice. First, there is little available data about how performance bugs are discovered, reported, and fixed in practice. Without such data, it is difficult to build tools that help developers effectively address performance bugs, beyond identifying slow code. For example, we find that performance bugs are more difficult to fix than non-performance bugs; therefore, developers may appreciate a tool that prioritizes performance bugs based on the predicted patch size—the smaller the patches, the better. Second, a robust end-to-end testing process requires more than just fault localization, which is what profilers are effectively doing. For example, for non-performance bugs, it is a standard and basic procedure to use automated oracles (e.g., crashes or assertions) to

[1] "Performance bug" is a well accepted term in some communities, e.g., Mozilla Bugzilla defines it as "A bug that affects speed or responsiveness" [12]. However, others prefer "performance problem" or "performance issue", because these problems differ from functional bugs. We take no position on this and use "performance bug" and "performance problem" interchangeably.

detect that a test has failed or has passed. However, for performance bugs, developers manually inspect the profiler output—and often times the source code—to infer if the test execution has triggered a performance bug. As another example, when fixing performance bugs, developers face a difficult trade-off between the advantages and the disadvantages caused by the fix. Finding and automatically repairing performance bugs that are likely to be fixed is of great value to developers.

We address the first problem in Chapter 2 and the second problem in Chapter 3 and Chapter 4. The work in Chapter 2 was published in [62], the work in Chapter 3 was published in [64], and the work in Chapter 4 is work in progress.

1.1 Discovering, Reporting, and Fixing Performance Bugs

Both industry and the research community have spent great effort on addressing performance bugs. For example, many projects have performance tests, bug tracking systems have special labels for performance bugs [4], and operating systems such as Windows 7 provide built-in support for tracking operating system performance [26]. In addition, many techniques are proposed recently to detect various types of performance bugs [8, 13, 18, 25, 29, 31, 40, 66, 76, 84, 86, 93, 94].

To understand the effectiveness of these techniques and design new effective techniques for addressing performance bugs, we need a deep understanding of performance bugs. A few recent papers [34, 91, 92] study various aspects of performance bugs, such as the root causes, bug types, and bug sources, which provides guidance and inspiration for researchers and practitioners. However, several research questions have not been studied at all or in depth, and answers to these questions can guide the design of techniques and tools for addressing performance bugs in the following ways:

- Based on maxims such as "premature optimization is the root of all evil" [43], it is widely believed that performance bugs *greatly differ* from non-performance bugs, and that patching performance bugs carries a much greater risk of introducing new functional bugs. A natural question to ask is compared to fixing non-performance bugs, whether fixing performance bugs is

indeed more likely to introduce new functional bugs. If fixing performance bugs is not more error-prone than fixing non-performance bugs, then developers may not need to be over-concerned about fixing performance bugs.

- Different from most non-performance bugs, whose unexpected behaviors are clearly defined, e.g., crashes, the definition of performance bugs is vague, e.g., how slow should a computation be to qualify.

  Therefore, are performance bugs more difficult to fix than non-performance bugs? For example, are performance bug patches bigger? Do performance bugs take longer to fix? Do more developers and users discuss how to fix a performance bug in a bug report? Many techniques are proposed to help developers fix bugs [28, 35, 36, 45], typically with a focus on non-performance bugs. If performance bugs are more difficult to fix, we may need more support to help developers fix them.

- Since the definitions of expected and unexpected behaviors for performance bugs are vague compared to those of non-performance bugs, are performance bugs less likely to be discovered through the observation of unexpected behaviors than non-performance bugs? Are performance bugs discovered dominantly through profiling because many profiling tools are available and used [57, 67, 90]? If performance bugs are less likely to be discovered through the observation of unexpected behaviors compared to non-performance bugs, or performance bugs are rarely discovered through profiling, then it is important for researchers and tool builders to understand the reasons behind the limited utilization of these techniques and address the relevant issues. If developers resort to other approaches to discover performance bugs, we may want to provide more support for those approaches to help developers detect performance bugs.

To answer these and related questions, we conduct a comprehensive study to compare performance and non-performance bugs regarding how they are discovered, reported, and fixed. Specifically, we manually inspect and compare 210 performance bugs and 210 non-performance bugs from three mature code bases: Eclipse Java Development Tools (JDT), Eclipse Standard Widget

Toolkit (SWT), and the Mozilla project. For questions where our analysis can be automated, we study an additional 13,840 non-performance bugs. However, identifying performance bugs requires manual inspection even when the analysis of the bug report and patches can be automated. Therefore, we do not increase the number of performance bugs for the automated experiments. The manual effort needed to study more performance bugs is an inherent limitation of our study and any similar study (details in Section 2.1.1). Nonetheless, the lessons learned from comparing these bugs should provide a good initial comparison between performance and non-performance bugs on discovering, reporting, and fixing them.

Chapter 2 answers the following research questions (RQ):

- **RQ1: Which is more likely to introduce new functional bugs: fixing performance bugs or fixing non-performance bugs?** It often takes multiple patches to completely fix a bug [68, 87], because (1) the initial patch may not completely fix the bug, (2) a patch may introduce a new functional bug (i.e., a bug that affects program's correct behavior) that requires additional patches to fix [87], and (3) the initial patch may need cosmetic changes or to be back-ported to other software releases. We refer to the first patch as the *initial patch*, and all subsequent patches for the same bug as *supplementary patches* following the terms used by Park et al. [68].

  Since it is commonly believed that patching performance bugs carries a greater risk of introducing new functional bugs, we want to identify the percentage of performance bugs whose patches introduce new functional bugs, and compare it against the percentage of non-performance bugs whose patches introduce new functional bugs. Therefore, we are only concerned with (2). Our results show that patching only 3.4–16.7% of performance bugs introduces new functional bugs, while patching 3.4–8.2% of non-performance bugs introduces new functional bugs. The differences are small and mostly statistically insignificant, which suggests that fixing performance bugs is about as error-prone as fixing non-performance bugs, indicating that developers may not need to be over-concerned about fixing performance bugs.

- **RQ2: Is fixing performance bugs more difficult than fixing non-performance bugs?** Compared to non-performance bugs, performance bugs consistently need more time to fix, more fix

attempts, more developers involved, and more time from the first to the last fix attempt. In addition, both the initial patches and the supplementary patches are considerably larger for performance bugs than for non-performance bugs. Furthermore, supplementary patches are less likely to be clones of an initial patch for fixing performance bugs, suggesting it is less likely that developers can use clone detection tools to find similar buggy locations to completely fix the bugs. These results show that performance bugs are probably more difficult to fix than non-performance bugs. While the current effort on helping developers fix bugs focuses on fixing non-performance bugs [28, 35, 36, 45], more support to help developers fix performance bugs is needed.

- **RQ3: How are performance bugs discovered and reported in comparison to non-performance bugs?** While the majority (84.5–94.5%) of non-performance bugs are discovered because users or developers observed their unexpected behaviors, e.g., system crashes, a much smaller percentage of performance bugs (30.2–49.2%) are discovered through the observation of unexpected program behaviors. Instead, a large percentage of performance bugs (33.9–57.3%) are discovered through *reasoning about code*. In addition, using a performance profiler amounts to only 5.5–10.4% of reported performance bugs. Since developers resort to code reasoning to discover performance bugs, we may want to provide more support to help developers perform code reasoning for discovering performance bugs. In addition, it is beneficial to have better profiling techniques, and better test oracles to help developers discover performance bugs through the observation of unexpected behaviors.

1.2 TODDLER: Detecting Performance Bugs via Similar Memory-Access Patterns

A key reason why performance bugs easily escape to production is that testing for performance bugs cannot use the well established process of testing for functional bugs with *automated oracles*.

An automated oracle detects if a test triggers a (functional or performance) bug, in which case the developer needs to inspect the test. To test for functional bugs, developers usually follow three steps: (1) write as many and as diverse tests as allowed by the testing budget, (2) run these tests and use automated oracles (e.g., crashes or assertions) to find which tests fail, and (3) inspect *only* the failing tests. To test for performance bugs, developers typically write a small number of tests, use a profiler to localize code regions that take a lot of time to execute, and then reason whether these regions can be optimized and if the effort spent for optimizing (time, added code complexity) is worth the potential speed gain (which may be difficult to ascertain before actually performing the optimization). In contrast to functional bugs, the lack of reliable automated oracles for performance bugs means that developers cannot easily find out which tests fail, as in step (2). As a result, because developers need to inspect all tests/profiles in step (3), they can use only a small number of performance tests in step (1). In summary, developers follow the current process of testing for performance bugs not because it has advantages, but because developers have no reliable alternatives.

An automated oracle for performance bugs would enable developers to test for performance bugs using the well established process of testing for functional bugs. Unfortunately, profilers cannot be used as effective oracles for three reasons. First, profilers give a *report for each test*, thus running many tests results in many reports, not just a few failing tests as for a typical functional oracle. Second, profilers may miss a performance bug *even when it is executed*: if the buggy code is not slow compared to the rest of that execution, it is not ranked high by the profiler and is likely ignored by the developer. Many performance bugs manifest by significantly degrading performance only for particular input conditions [31, 34, 38, 92], and the profiled inputs cannot cover all possible conditions. Third, profilers report what *takes* time but not what *wastes* time, i.e., they do not distinguish truly necessary (albeit expensive) work from the likely unnecessary computation. In other words, profilers are highly useful when the developer wants to *localize* a slow code region but are not effective when the developer needs to *decide* if a test likely exposes a performance bug and thus needs further inspection.

Chapter 3 presents TODDLER, a novel *oracle* for performance bugs. In brief, TODDLER reports tests that execute loops whose computation is repetitive and very similar across iterations. The intuition is that such loops are likely performance bugs that waste time: because the work is repetitive and similar, it could be done faster. We designed TODDLER based on two observations about performance bugs. First, many severe performance bugs (over 50% in the study in Section 4.1) are contained by nested loops: if an inefficient code region is executed outside of a nested loop, then the inefficiency itself needs to be very severe (e.g., slow I/O) for the code region to have a real impact on the overall program performance. Second, wasted computation is often reflected by repetitive and partially similar memory accesses across loop iterations: if a group of instructions repeatedly accesses similar memory values, then those instructions probably compute similar results.

We implemented a full-blown TODDLER tool for Java and a simple prototype for C/C++. Our experiments with 11 previously known, real-world performance bugs from 9 Java projects show that TODDLER is able to find all these bugs. Our C/C++ prototype also finds 6 previously known bugs in GCC, Mozilla, and MySQL. Moreover, using TODDLER helped us identify *42 new real-world bugs* in six popular Java projects: Ant, Google Core Libraries, JUnit, Apache Collections, JDK, and JFreeChart. Based on our reports, developers so far have fixed 10 bugs and confirmed 6 more as real bugs, and the Apache Collections developers even invited the author of this dissertation to become a project committer. Our bug reports are linked from `http://mir.cs.illinois.edu/toddler`.

1.3 LULLABY: Detecting and Fixing Performance Bugs that Have Non-intrusive Fixes

Several techniques [8, 13, 18, 25, 29, 31, 40, 66, 76, 84, 93, 94] have been proposed to help detect various types of performance bugs. However, there are still many performance bugs that cannot be detected by existing techniques. Furthermore, a crucial and practical aspect of performance bugs

has not received the attention it deserves: *how likely are developers to fix a performance bug?*

In practice, when developers decide if they should fix a performance bug, developers face a difficult choice between the potential drawbacks and the potential benefits of fixing the bug.

On one hand, similar to fixing functional bugs, fixing performance bugs can have drawbacks. First, fixing performance bugs may introduce severe functional bugs, which may lead to program crashes or data losses. The risk of having such negative and noticeable effects can make developers very cautious about improving performance. Second, fixing performance bugs may break good software engineering practices, making the code difficult to read, maintain, and evolve. For example, fixing performance bugs may require breaking encapsulation, code cloning, or specialization. Third, fixing performance bugs takes time and effort, especially if the fix involves several software modules or requires a complex implementation. Fourth, fixing performance bugs, which manifest for some inputs, may slow down other code, for some other inputs, and developers must decide which of these slowdowns—the slowdown caused by the performance bug for some inputs, or the slowdown caused by the fix for some other inputs—is preferable.

On the other hand, fixing performance bugs has benefits, i.e., it speeds up code. However, unlike fixing functional bugs, the benefits of fixing a performance bug *are often difficult to assess accurately*, especially when the fix is complex. First, the speedup offered by the fix depends on the input, and many inputs may not be sped up at all, because performance bugs manifest only for certain inputs. Therefore, developers need to estimate which inputs have which speedups, and how frequent or important these inputs are in practice. Second, the exact speedup offered by the fix for an input is difficult to estimate without executing the code, and speedups of orders of magnitude—i.e., speedups for which accurate estimates are not necessary—are rare. Unfortunately, developers often have access to only a few real-world inputs triggering a bug—or none at all, if the bug was detected during development using benchmarks, static tools, or code inspection—and can find it difficult to estimate the expected speedup for the rest of the real-world inputs that may trigger the bug.

In practice, developers fix performance bugs when the benefits outweigh the drawbacks. In

particular, developers are likely to fix performance bugs that have simple and non-intrusive fixes. Such fixes are unlikely to introduce new functional bugs, do not increase code complexity and maintenance costs, are easy to understand and implement, and are unlikely to degrade performance for other inputs. In other words, the choice between benefits and drawbacks is made easy for developers: because the fixes are simple and non-intrusive, fixing the bugs can only bring benefits.

Chapter 4 makes the following contributions:

New problem: Instead of detecting performance bugs that developers may decide not to fix, this chapter proposes a novel problem: *detecting performance bugs that developers are likely to fix*. Following the above discussion, we propose to detect performance bugs whose fixes clearly provide more benefits than drawbacks.

A family of performance bugs: To address this problem, this chapter identifies a family of performance bugs that developers are likely to fix. Every bug in this family is associated with a loop[2] and a condition. When the condition becomes true during the loop execution, all the remaining computation performed by the loop is wasted. In the extreme case when the condition is true at the start of the loop execution, the entire loop computation is wasted. Developers typically fix performance bugs in this family because (1) these performance bugs waste computation in loops, and (2) these performance bugs have simple and non-intrusive fixes: when some condition becomes true during the loop execution, just break out of the loop. Typically, these performance bugs are fixed by adding a single line of code inside the loop, i.e., `if (cond) break`, which we call an *CondBreak fix*. We call `cond` a *L-Break condition*.

Technique: This chapter proposes LULLABY, a novel static technique for detecting performance bugs that have CondBreak fixes. LULLABY takes as input a program and outputs loops that can be fixed by CondBreak fixes, together with a potential fix for each buggy loop. The fixes proposed by LULLABY can be directly applied to source-code and are easy to read by developers. LULLABY works on intermediate code representation and analyzes each loop in five steps. First,

[2]We focus on loops because most computation time is spent inside loops and most performance bugs involve loops [30, 34, 64, 82, 93].

LULLABY identifies the loop instructions that may produce results visible after the loop terminates. Second, for each such instruction, LULLABY detects the condition under which the instruction can be skipped for the remainder of the loop without changing the program outcome. We call this condition an *I-Break condition*, similarly to the L-Break condition described earlier for the entire loop. Third, LULLABY checks if all instructions from step two can be skipped *simultaneously* without changing the program outcome, i.e., if all I-Break conditions can be satisfied simultaneously. The conjunction of all I-Break conditions is the L-Break condition. Fourth, LULLABY checks if the computation waste in the loop is not already avoided, i.e., if the loop does not already terminate when the L-Break condition is satisfied. If all the previous steps are successful, LULLABY reports a performance bug. Fifth, LULLABY generates a fix for the performance bug. The fix has the basic format `if (cond) break`, where `cond` is the L-Break condition LULLABY computed in the third step.

Automatic fix generation: Automated bug fixing is challenging in general [28], but LULLABY is able to automatically generate fixes for most bugs it finds because LULLABY takes advantage of two characteristics of the performance bugs it detects: (1) the bugs have CondBreak fixes, which can be inserted right at the start of the loop, thus avoiding complex interactions with the other loop code, and (2) the L-Break condition in a CondBreak fix is a relatively simple boolean expression, as described in Section 4.1 and Section 4.2.

Evaluation: We implement *two* LULLABY tools, one for Java, LULLABY-J, and one for C/C++, LULLABY-C. We evaluate LULLABY-J on 11 popular Java applications: Ant, Groovy, JMeter, Log4J, Lucene, PDFBox, Sling, Solr, Struts, Tika, and Tomcat. LULLABY-J found *61 new real-world performance bugs* of which *51 bugs have already been fixed by developers*. We evaluate LULLABY-C on 4 widely used C/C++ desktop and server applications: Chromium, GCC, Mozilla, and MySQL. LULLABY-C found *89 new real-world performance bugs* of which *65 bugs have already been fixed by developers*. Of the bugs not yet fixed, 7 bugs are confirmed and still under consideration by developers, and 16 bugs are still open. 7 bugs were not fixed because they are in deprecated code, old code, test code, or auxiliary projects. *Only 4 bugs* were not fixed because

developers considered that the bugs have small performance impact or that the fixes make code more difficult to read. LULLABY has few false positives, 19 for LULLABY-J and 4 for LULLABY-C, mostly due to limitations in the static analysis framework we use. Out of 150 bugs, LULLABY successfully generates fixes for 149 bugs. For one bug, LULLABY-J did not generate a fix due to a limitation in the static analysis framework.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 presents our study of real-world performance bugs and how they compare with non-performance bugs. Chapter 3 presents TODDLER, an automated oracle for performance bugs that enables developers to test for performance bugs using the well established process of testing for functional bugs. Chapter 4 presents LULLABY, a technique for detecting performance bugs that developers are likely to fix. Chapter 5 presents related work, and Chapter 6 discusses potential future projects and concludes the dissertation.

Chapter 2

Discovering, Reporting, and Fixing Performance Bugs

This chapter presents the contributions of our performance bug study, which was undertaken with the goal of understanding performance bugs. This chapter is organized as follows. Section 2.1 presents the study methodology, Section 2.2 presents the study results and findings, Section 2.3 discusses the threats to validity, and Section 2.4 summarizes our study.

2.1 Experimental Methods

2.1.1 Collection of Bugs and Patches

We choose three large, mature, and popular projects as subjects to study: Eclipse Java Development Tools (JDT), Eclipse Standard Widget Toolkit (SWT), and Mozilla.

We reuse the bugs studied by Park et al. [68], because answering some of our research questions requires distinguishing between bugs that were fixed correctly on the first attempt and bugs that involve several attempts to be fully fixed, which was studied by that work. However, that work does not distinguish performance bugs from non-performance bugs, and therefore does not answer the research questions addressed in this chapter. To identify bug-fixing commits and their corresponding supplementary patches, Park et al. search commit logs for bug report IDs from bug databases. If a bug report ID was found in a commit log, they consider the commit a bug-fixing commit for the bug report. If multiple commits contain the same bug report ID, then the first commit is the initial commit, and all subsequent commits are supplementary patches for the bug report. While this approach can miss some patches that are related to the bug (if the commit messages

Table 2.1: Characteristics of the study subjects.

	JDT	SWT	Mozilla	Sum
First/Last Commit	2001–2009	2001–2010	1998–2011	/
Bug Study Period	2004–2006	2004–2006	2003–2005	/
Lines of Code	262,332	266,870	913,130	/
Number of Authors	18	27	754	/
Number of Commits	17,009	21,530	261,630	/
I-Perf (Perf. bugs inspected manually)	55	59	96	210
I-NonPerf (Non-perf. bugs inspected manually)	79	73	58	210
A-NonPerf (Non-perf. bugs processed automatically)	1,781	1,223	11,046	14,050

for these patches do not have the bug ID), the approach still provides a highly useful dataset that helped Park et al. answer the questions in their study and also allows us to answer our research questions.

Following Park et al. [68], we study bugs reported between 2004–2006 for JDT and SWT, and 2003–2005 for Mozilla, but still include patches for these bugs beyond the time periods above (until 2009, 2010, 2011 for JDT, SWT, and Mozilla, respectively), to ensure the studied bugs are completely resolved and no additional supplementary patches for these bugs are likely to appear in the future.

Table 2.1 summarizes the characteristics of the studied subjects: time period between the first and last commits in the repository, time period in which the studied bugs were reported (as described above), lines of code and number of authors at the end of the studied period, and the total number of commits.

Three Bug Sets: The last three rows in Table 2.1 give the number of performance and non-performance bugs studied in this chapter. We divide the bugs in three different bug sets—*I-Perf, I-NonPerf, and A-NonPerf*—and use the appropriate bug sets depending on the research questions we answer. Bug set **I-Perf** contains 210 bugs that we manually identified to be performance bugs (Section 2.1.2 describes our manual inspection process). Bug set **I-NonPerf** contains 210 bugs

that we manually identified to be non-performance bugs, which we use only in experiments that need manual inspection. Otherwise, we use bug set **A-NonPerf**, which contains all the bugs except the bugs in I-Perf (i.e., A-NonPerf includes also bugs that do not contain any of the keywords that are used to search for performance bugs). We consider these bugs to be all the non-performance bugs in our study. These bugs are not identified through manual inspection. The vast majority of these bugs are non-performance bugs, although a few of them may be performance bugs (details in Section 2.3). Note that unlike A-NonPerf for non-performance bugs, we do not have an A-Perf category for performance bugs. The reason is that only a small percentage of reported bugs are performance bugs, and identifying them requires manual inspection. Therefore, we still use I-Perf for experiments that can be automated. We explain why and how we collect the three sets in Section 2.1.2 and discuss the threats to this method in Section 2.3.

2.1.2 Manual Inspection of Bugs and Patches

Identifying Performance Bugs: To assign a bug to I-Perf, we use an approach similar to other studies of concurrency, security, and performance bugs [27, 34, 48, 91]. We first identify the bug reports that contain a performance-related keyword ("performance", "slow" , "speed", "latency", and "throughput") in the bug description, bug summary, or the discussion developers had while solving the bug. For JDT, SWT, and Mozilla, this step finds 135, 108, and 1,101 bugs, respectively. We manually inspect all the 135 and 108 bugs for JDT and SWT, and we randomly sample 450 of the 1,101 bugs for Mozilla. During this manual inspection, we read the bug description and the discussion developers had while fixing the bug, and decide if the inspected bug is a performance bug or not. To ensure the correctness of our results, this manual inspection step is performed independently by two researchers. For the bugs where the results from the two inspections differ, the researchers discuss to reach a consensus. In this way, we identified a total of 210 performance bugs—55, 59, and 96 in JDT, SWT, and Mozilla, respectively.

Identifying Non-Performance Bugs: To assign a bug to I-NonPerf, we randomly select bugs from A-NonPerf, manually inspect them, and keep in I-NonPerf only bugs that we manually verify

as non-performance bugs.

Classifying Supplementary Patches By Purposes: To answer RQ1, we need to know which supplementary patches fix new functional bugs that were introduced by other patches. We manually inspect the supplementary patches in our data sets I-Perf and I-NonPerf to classify the supplementary patches into five categories based on their purposes, i.e., fixing new functional bugs introduced by other patches, improving the performance or completing the initial patch, a combination of these two purposes, making only a code formatting change, or only applying the initial patch or its variant to a different branch. To do so, we manually examine the supplementary patches, the commit logs for these patches, and the bug reports.

Identifying Mechanisms to Detect Bugs and Information Provided With a Bug Report: To answer RQ3, we manually examine bug reports to classify bug reports according to how the bugs are discovered into four categories: discovered through code reasoning, through observation of unexpected behaviors, through the failure of regression tests, or through using a profiler. In addition, we classify bug reports in a different dimension based on what information is provided to help reproduce and fix bugs into three categories: inputs are provided, steps to reproduce are provided but inputs are not provided, or neither inputs nor steps to reproduce are provided.

2.1.3 Statistical Tests

We work with the statistical consulting service provided by the University of Waterloo to use the proper statistical tests to understand whether there is a statistically significant difference between two values that we want to compare.

We report statistical measures when applicable. For example, to answer part of our RQ1, we compare the proportion of performance bugs that require supplementary patches to the proportion of non-performance bugs that require supplementary patches. Since Mozilla has many bugs, we randomly sample some bugs to be manually inspected (I-Perf and I-NonPerf). We want to understand whether the proportions we obtain from the sample are likely under the null hypothesis. We set the null hypothesis to be "the probability of a performance bug causing supplementary patches

is the same as the probability of a non-performance bug causing supplementary patches".

We model the experiment as a coin-flip experiment. For example, given a bug, it can be either a performance bug (head) or a non-performance bug (tail). We choose to use the Fisher's exact test in this situation because the Fisher's exact test does not require the data to follow a normal distribution, and is appropriate even if the sample size is small. For other experiments whose data are ordinal, such as the number of supplementary patches, we apply Mann-Whitney U-test. We choose U-test over t-test because the t-test assumes a normal distribution while the U-test does not. At a 95% confidence level, we reject the null hypothesis if the p-value is smaller than 0.05. A p-value greater than 0.05 indicates that we do not find strong enough evidence to reject the null hypothesis.

For experiments on JDT's and SWT's I-Perf and A-NonPerf data sets, no statistical test is needed to extend the results from the sample to the entire population because we examined the entire population of bugs in the given period of time. Any difference is a factual difference on the studied population. In the tables in the rest of this chapter, we do not show the p-value column for those cases or use "/" to denote the irrelevant cells. Since we use keyword search first to find our studied population (similar to prior work [27,34,48,91]), which is not a random sample, statistical measures such as t-test, U-test, and p-values (which all assume random samples) do not directly generalize our results beyond the studied population. Given the small percentage of performance bugs, it is prohibitively expensive to randomly sample bug reports and still find enough performance bugs for a representative study. Keyword search is our best effort, as commonly done in previous related studies. Similar to prior studies [27, 68, 71], the studied time period and projects are not randomly selected. We discuss these threats further in Section 2.3.

Due to the space constraints, we only explain one null hypothesis in detail. For part of RQ1, we want to check if the proportion of performance bugs that are multi-patch are the same as the proportion of non-performance bugs that are multi-patch. Here multi-patch bugs are bugs that require supplementary fixes. A naive approach is to formulate the null hypothesis about the conditional probabilities of these types of bugs as $P(Multi\text{-}Patch|Perf) = P(Multi\text{-}Patch|NPerf)$. However,

directly evaluating those two probabilities requires a good estimate of the number of performance and non-performance bugs. Since we sample bugs to determine performance and non-performance bugs, we do not know their precise numbers, which could introduce errors. Fortunately, we can avoid such errors by rewriting this hypothesis into a different form: $P(Perf|Multi\text{-}Patch) = P(Perf|Uni\text{-}Patch)$. This form has two desired properties. First, this form is mathematically equivalent to the original hypothesis. The proof follows from the basic axioms of probability. Second, this form avoids the errors described above because we know the exact number of Uni-Patch and Multi-Patch bugs in the dataset.

2.2 Results

2.2.1 RQ1: Which Is More Likely to Introduce New Functional Bugs: Fixing Performance Bugs or Fixing Non-performance Bugs?

To answer our RQ1, we first (1) determine whether fixing a bug requires supplementary patches, and then (2) manually inspect whether the supplementary patches fix new functional bugs (i.e., new functional bugs that were introduced by the bug's patches). These steps allow us to compare the percentage of performance bugs whose patches introduce new functional bugs against the percentage of non-performance bugs whose patches introduce new functional bugs.

For step (1), we split performance bugs and non-performance bugs in two categories: *uni-patch bugs*, which are fixed with only one patch, and *multi-patch bugs*, which are fixed with two or more patches, i.e., bugs that developers did not fix correctly or fully in the first attempt. This method was used by Park et al. [68] to identify bug patches that introduce new bugs. The intuition is that the existence of supplementary patches indicates that the initial patch was either incomplete (it did not fully fix the bug) or incorrect (it introduced new bugs that needed to be fixed).

For step (2), we classify the multi-patch bugs into the following five disjoint categories according to the goals of their supplementary patches. When developers fix a bug (referred to as

the original bug for clarity), if they introduce a *new* functional bug during this process, and the supplementary patches fix the *new* functional bug introduced, then the original bug belongs to the category *FixNewFunc*. Since developers may use multiple patches to completely fix the original bug, we consider new functional bugs introduced by all these patches that fix the original bug. These patches are relevant to RQ1 because we want to study how likely it is to introduce new functional bugs when fixing the original (performance or non-performance) bugs. *FixOld&Perf* represents the bugs whose supplementary patches complete the fix in the initial patch or improve performance on top of the performance gain from the initial patch. This category only includes bugs whose supplementary patches do not fix new functional bugs. Since a bug can have multiple patches, some of which fix a new functional bug, and some of which complete the initial fix or improve performance, we use *Both* to denote these bugs. *Format* represents the bugs whose patches perform *only* cosmetic changes, such as adding comments. *To Branch* represents the bugs whose patches *only* apply a fix similar to the initial patch, but to a different branch.

To answer RQ1, we take the multi-patch performance bugs from step (1) and calculate what percentage belong to *FixNewFunc* and *Both*, both of which are bugs whose patches introduce new functional bugs. Similarly, we calculate the same percentage for non-performance bugs. Table 2.2 shows the results. These numbers show that only a small percentage (3.4–16.7%) of performance bugs have patches that introduce new functional bugs. These results also show some differences between performance and non-performance bugs, i.e., fixing performance bugs is less likely to introduce new functional bugs than fixing non-performance bugs for SWT, and fixing performance bugs is more likely to introduce new functional bugs than fixing non-performance bugs for JDT and Mozilla.

We then performed a careful statistical analysis to determine how significant these differences are. The null hypothesis is "fixing performance bugs is as likely to introduce new functional bugs as fixing non-performance bugs", and column p-val in Table 2.2 shows the p-values. For JDT and SWT, the p-values are greater than 0.05, indicating that there is no statistically strong evidence to show that fixing performance bugs and fixing non-performance bugs are different

in terms of introducing new functional bugs. For Mozilla, the difference between performance and non-performance bugs is statistically significant, but it is small. These results show that the common belief that patching performance bugs carries a greater risk of introducing new functional bugs may not be true. Therefore, developers may not need to be over-concerned about fixing performance bugs.

Table 2.2: Percentages of performance and non-performance bugs whose patches introduce new functional bugs. This table uses the I-Perf and I-NonPerf datasets.

App	NPerf (%)	Perf (%)	p-val
JDT	6.3	7.3	>0.99
SWT	8.2	3.4	0.3
Mozilla	3.4	16.7	0.02

Fixing performance bugs is about as likely to introduce new functional bugs as fixing non-performance bugs.

Below we present the detailed results of our steps (1) and (2). Table 2.3 shows our results comparing the ratios of performance and non-performance bugs (columns *Perf* and *NPerf*) in the uni-patch and multi-patch categories (columns *Uni-Patch* and *Multi-Patch*); columns # and % give the number and percentage of bugs. About 31% (25.4% to 36.4%) of performance bugs require additional patches after the initial patch, whereas about 27% (22.3% to 32.6%) of non-performance bugs require additional patches. While the percentage for performance bugs is consistently higher than for non-performance bugs, the differences are small. Therefore for practical purposes, ***fixing performance bugs is about as likely to require supplementary patches as fixing non-performance bugs***.

For the multi-patch performance bugs, we want to know why these bugs need supplementary patches. For example, are performance bugs so difficult to fix that their patches introduce new functional bugs? Or, do the supplementary patches add more performance improvements on top of the initial patch? To understand the reason for supplementary patches, for each performance bug, we manually analyze its supplementary patches (one bug may have more than one supplementary

Table 2.3: Performance and non-performance bugs that need (*Multi-Patch*) or do not need (*Uni-Patch*) additional fixing after the initial patch. This table uses the I-Perf and A-NonPerf datasets.

App	Uni-Patch				Multi-Patch			
	NPerf		Perf		NPerf		Perf	
	#	%	#	%	#	%	#	%
JDT	1,383	77.6	41	74.5	398	22.3	14	25.4
SWT	928	75.8	39	66.1	295	24.1	20	33.9
Mozilla	7,443	67.3	61	63.5	3,603	32.6	35	36.4

patch; Section 2.2.2 gives quantitative data for the number of supplementary patches), the commit logs for these patches, and the bug report.

We classify multi-patch bugs into five categories based on the purposes of their supplementary patches. Table 2.4 shows the percentage of multi-patch performance bugs and non-performance bugs that belong to the five categories. Perhaps surprisingly, the majority of performance bugs needs supplementary patches not because their patches introduced a new functional bug that needed to be fixed, but rather because the developers wanted to further improve performance, in addition to the improvements already made in the initial patch. For example, for SWT, only 5% of the multi-patch performance bugs have supplementary patches that fix new functional bugs introduced by their patches, while 75% of the multi-patch performance bugs have supplementary patches that further improve performance or complete the initial patch. JDT and Mozilla have similar results (21.4% and 64.3% for JDT, 22.9% and 31.4% for Mozilla). For a relatively large fraction of performance bugs of up to 22.9%, the supplementary patches only port the initial patch to older released branches.

Bug and Patch Examples: Figure 2.1 shows an example supplementary patch (for the Mozilla 240934 bug) that further improves the performance gained by the initial patch. The high level fix idea for Mozilla 240934 is to search using a hashtable instead of performing a linear search over an array. The initial patch incorporates a hashtable in the code, changing 348 lines of code, a relatively large patch. Later, the developer realizes that the hashtable can use a better hash method,

Table 2.4: Why do developers need supplementary patches? This table uses the multi-patch bugs from I-Perf and I-NonPerf. The values represent the percentage of bugs in each category out of the multi-patch bugs.

Why?	JDT (%)		SWT (%)		Mozilla (%)	
	NPerf	Perf	NPerf	Perf	NPerf	Perf
FixNewFunc	22.2	21.4	11.1	5.0	10.5	22.9
FixOld&Perf	44.4	64.3	61.1	75.0	42.1	31.4
Both	5.6	7.1	22.2	5.0	0.0	22.9
Format	5.6	0.0	5.6	0.0	5.3	0.0
To Branch	22.2	7.1	0.0	15.0	42.1	22.9

and implements this better hash method in the second patch, as shown in Figure 2.1. In other words, performance is already improved by the initial patch, and the supplementary patch just adds to the initial improvement.

```
1 Index: trunk/mozilla/layout/html/base/src/nsPresShell.cpp
2 ===================================================
3 @@ -1031,11 +1031,12 @@
4 ...
5 -     NS_PTR_TO_INT32(command->GetTarget()) ^
6 +     (NS_PTR_TO_INT32(command->GetTarget()) >> 2) ^
7 ...
```

Figure 2.1: A supplementary patch for Mozilla bug 240934, which further improves performance by using a better hash function, in addition to the improvement offered by the initial patch.

For some bugs (e.g., SWT 99524, JDT 89096, Mozilla 239358, and SWT 120721), the initial and supplementary patches are one high level fix. However, developers chose to commit this high level fix in several different patches (which became the initial and supplementary patches), either because the different patches represented logically different coding sub-tasks, or simply because the overall fix was large and the developer implemented it in several stages. For example, the patches for Mozilla 239358 total over 900 lines of code, and it appears that the developer implemented and committed the different patches in several stages.

The patches for performance bugs can indeed introduce new functional bugs. Figure 2.2 shows an example supplementary patch, that fixes the bug inserted by the initial patch for the performance bug Mozilla 221361. The new functional bug created by the initial patch was found and reported

(as Mozilla 270297) one year after it was introduced. The initial patch for Mozilla 221361 changed 36 lines of code, among them several lines doing pattern matching on strings, similar to line 5 in Figure 2.2. Among so many changes, the developer got one pattern wrong (line 5 in Figure 2.2), which makes Firebird build wrong URLs.

```
1 Index: trunk/mozilla/browser/base/content/browser.js
2 ===================================================
3 @@ -4801,7 +4801,8 @@
4 ...
5 -         searchStr = searchStr.replace(/\s*(.*?)\s*$/, "$1");
6 +         searchStr = searchStr.replace(/^\s+/, "");
7 +         searchStr = searchStr.replace(/\s+$/, "");
8 ...
```

Figure 2.2: Supplementary patch that fixes a new functional bug (Mozilla 270297) created by the initial patch of a performance bug (Mozilla 221361).

The majority of performance bugs have supplementary patches that either improve the performance gains offered by the initial patch or complete the implementation of the initial patch. Relatively few performance bugs have supplementary patches that fix new functional bugs introduced by other patches of the same original bug.

2.2.2 RQ2: Is Fixing Performance Bugs More Difficult Than Fixing Non-performance Bugs?

This section studies whether fixing performance bugs is more difficult than fixing non-performance bugs. While it is hard to define and quantify the "difficulty" of fixing a bug, we study a wide spectrum of aspects of fixing performance and non-performance bugs to provide some understanding toward this end. Such information can help developers prioritize the types of bugs to fix and estimate the resources needed.

We first investigate the average time necessary to fully fix bugs, and the number of developers and users involved in the discussion that lead to the final fix. Second, for the bugs that require more than one fix attempt (i.e., the multi-patch bugs) and are thus more difficult to fully fix, we

present the total number of fix attempts and the time from the initial (insufficient) patch to the last patch. These numbers approximate the extra effort needed to fully fix multi-patch bugs. Third, we study the size of the initial patch, and, if the initial patch did not fully fix the bug, the size of the supplementary patches. These numbers approximate the effort required to patch the bug and the complexity added to the code. Fourth, we investigate if clone detection can help developers discover similar buggy locations and fix incomplete initial patches that need additional fixing.

Table 2.5 shows the average time (in days) that took to resolve a bug, from when it was first reported, to when it was closed. Performance bugs usually take more time than non-performance bugs to be resolved, e.g., about 75 more days on average for SWT and Mozilla.

Table 2.5: Time necessary to fully resolve a bug. This table uses the I-Perf and A-NonPerf datasets.

App	NPerf	Perf
JDT	126.4	123.3
SWT	201.0	275.9
Mozilla	655.8	730.8

Table 2.6 shows the average number of developers that took part in the discussion about how to fix the bug. Fixing performance bugs consistently involved more developers than fixing non-performance bugs. For example, SWT needs an average of 3.3 developers to fix a non-performance bug, but an average of 3.9 developers to fix a performance bug. The results are statistically significant for Mozilla because the p-value is less than 0.05. The differences are factual differences for JDT and SWT since we examine the entire population as explained in Section 2.1.3 (denoted by "/").

Table 2.6: Developers involved in fixing a bug. This table uses I-Perf and A-NonPerf. "/" indicates that p-values are not needed since we examine the entire population as explained in Section 2.1.3.

App	NPerf	Perf	p-val
JDT	3.7	3.9	/
SWT	3.3	3.9	/
Mozilla	5.2	6.5	8.0e-4

Figure 2.3 shows, for the bugs that are fixed more than once, the total number of patches required for each bug. Performance bugs consistently need more patches than non-performance bugs for all three applications. For example, fewer performance bugs need two or three patches than non-performance bugs. In other words, while performance bugs are not more likely to need supplementary patches than non-performance bugs (Section 2.2.1), the performance bugs that do need supplementary patches are more difficult to fix.

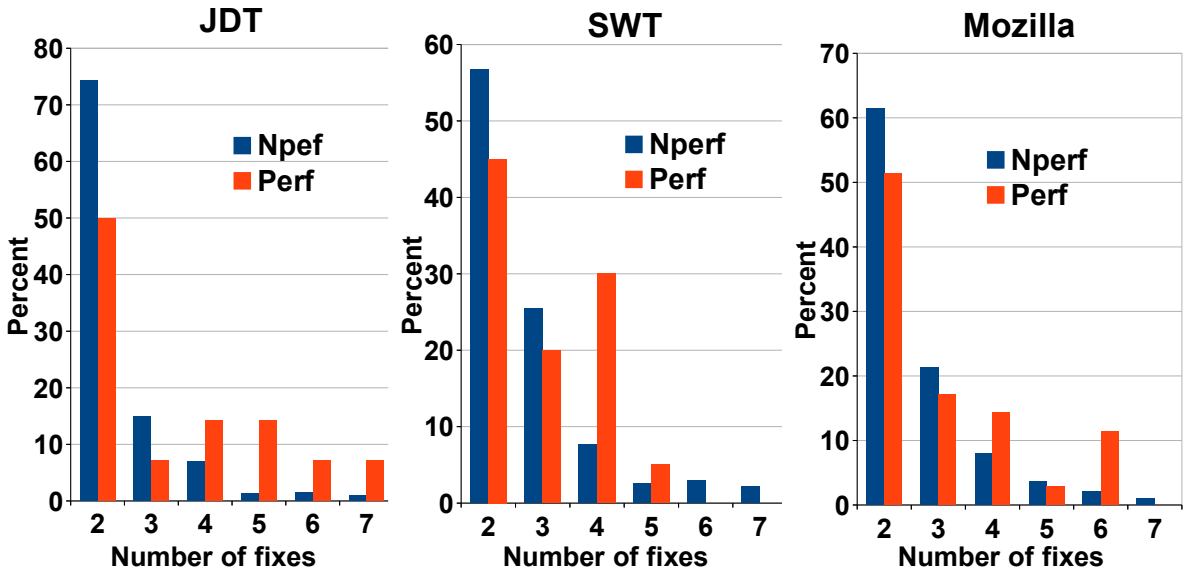

Figure 2.3: Number of times a bug is fixed. This figure uses the multi-patch bugs from I-Perf and A-NonPerf.

Figure 2.4 shows, for the bugs that are fixed more than once, the time between the initial patch and the last patch. These numbers show for how long the code was incompletely fixed, and indicate how difficult it was to fully fix the bug (note that these numbers apply only for multi-patch bugs and represent different data than the numbers in Table 2.5). For JDT and SWT, performance bugs need more time for the initial patch to be fully fixed compared to non-performance bugs; for Mozilla, performance bugs need less time than non-performance bugs.

Table 2.7 compares the size of the initial patch for performance and non-performance bugs. Column *Files* gives the average number of files changed, column *LOC* gives the average number

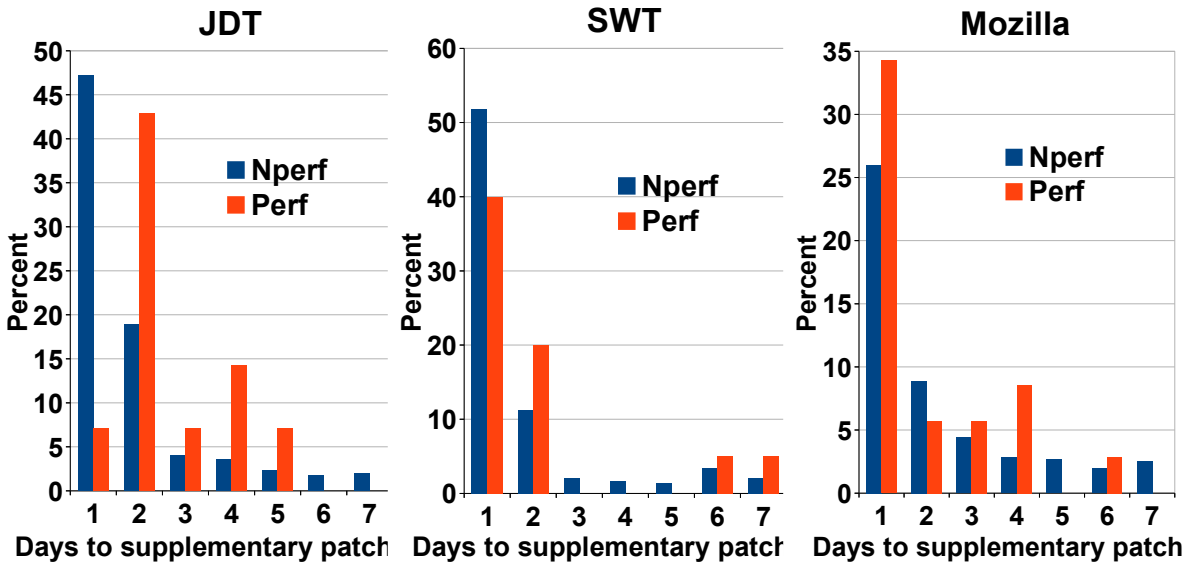

Figure 2.4: Time (number of days) between the first patch and the last supplementary patch. This figure uses the I-Perf and A-NonPerf datasets.

of lines of code changed, and column *Added LOC* gives the average number of code lines that are added, as a percent of the total number of lines changed. For all three applications, fixing performance bugs requires changing more files and more lines of code than fixing non-performance bugs. For example, for JDT, fixing non-performance bugs changes on average 2.8 files, while fixing performance bugs changes on average 4.4 files. The difference is even larger when considering the lines of code changed: double for JDT and Mozilla, and almost three times for SWT (44.2 lines for non-performance, and 131.4 for performance). Percentage-wise, performance bugs require slightly less added lines of code than non-performance bugs, but the differences are small, and in absolute numbers, performance bugs still add more lines of code than non-performance bugs. Overall, the initial patch is considerably larger for performance bugs than for non-performance bugs.

Table 2.8 compares, for performance and non-performance bugs which have an incomplete first patch, the size of the supplementary patches. Columns *Files*, *LOC*, and *Added LOC* give the average number of files changed, lines of code changed, and lines of code added, respectively. For JDT and Mozilla, the supplementary patches for performance bugs are considerably larger than those

Table 2.7: Size of initial patches. This table uses I-Perf and A-NonPerf.

App	Files			LOC			Added LOC%		
	NPerf	Perf	p-val	NPerf	Perf	p-val	NPerf	Perf	p-val
JDT	2.8	4.4	/	108.4	211.2	/	67.9	60.9	/
SWT	1.9	2.5	/	44.2	131.4	/	72.0	69.5	/
Mozilla	4.2	8.2	0.01	212.5	570.7	2.6e-7	62.9	58.4	0.02

for non-performance bugs by all three metrics; for SWT, they are slightly smaller. These numbers indicate that, even though performance bugs are not more likely to have incomplete patches than non-performance bugs (Section 2.2.1), when they do, the supplementary patches are much more complex than for non-performance bugs.

Table 2.8: Size of additional patches. This table uses the I-Perf and A-NonPerf datasets.

App	Files			LOC			Added LOC%		
	NPerf	Perf	p-val	NPerf	Perf	p-val	NPerf	Perf	p-val
JDT	3.0	4.0	/	96.9	222.0	/	62.0	66.2	/
SWT	3.6	3.4	/	179.6	141.0	/	76.8	72.9	/
Mozilla	6.6	11.5	0.07	343.5	480.1	0.01	64.1	65.3	0.12

Clone detection has been used to detect and fix bugs. We compare how this technique works for performance and non-performance bugs. Figure 2.5 gives the percentages of the supplementary patches that are either clones of the initial patches, backports of the initial patches, or neither. For JDT and SWT, performance bugs have a smaller percentage of clones of their initial patches than non-performance bugs, while for Mozilla, the percentages are about the same. This means that clone detection is probably less effective for helping developer fix performance bugs completely than fixing non-performance bugs completely, because clone detection tools are less likely to find buggy locations similar to what the initial patch fixes to help developers complete the fix.

Fixing performance bugs is more difficult than fixing non-performance bugs. First, performance bugs need more time to be fixed, more fix attempts, more developers involved, and more

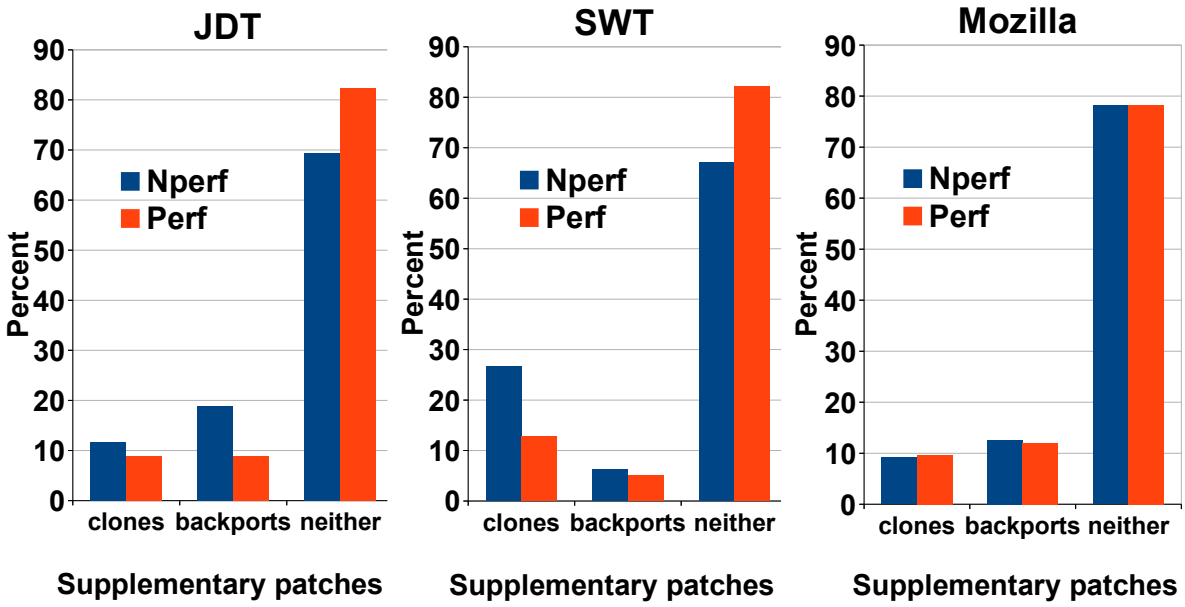

Figure 2.5: Supplementary patches that are clones, backports, or none of the two. This table uses the I-Perf and A-NonPerf datasets.

time from the first to the last fix attempt. In addition, both the initial and supplementary patches (if needed) for performance bugs are more complex than the patches for non-performance bugs. Furthermore, fewer performance bugs' supplementary patches are clones of the initial patches.

2.2.3 RQ3: How Are Performance Bugs Discovered and Reported in Comparison to Non-performance Bugs?

Since the expected and unexpected behaviors for performance bugs are less clearly defined than those of non-performance bugs, we want to know how users discover and report performance bugs, and how it compares to non-performance bugs. We investigate two aspects: (1) how the bug reporter discovered the bug, and (2) what input the bug reporter provided to help developers reproduce and fix the bug.

Table 2.9 shows how performance bugs were discovered compared to non-performance bugs.

Row *Reason* gives the percentages of bugs that were found through code inspection and reasoning. Row *Observe* shows the percentages of bugs that were found because users or developers observed the adverse effects of the bugs (e.g., program crashing and program running slow). Row *Test R.* presents the percentages of bugs that were found because a regression test failed. Row *Profiler* gives the percentages of bugs that were found because developers profiled the code.

Table 2.9: How are performance and non-performance bugs discovered? This table uses the I-Perf and I-NonPerf datasets.

How?	JDT (%)			SWT (%)			Mozilla (%)		
	NPerf	Perf	p-val	NPerf	Perf	p-val	NPerf	Perf	p-val
Reason	5.1	50.9	7.7e-10	4.1	33.9	1.2e-5	15.5	57.3	2.6e-7
Observe	91.1	36.4	1.4e-11	94.5	49.2	2.6e-9	84.5	30.2	2.9e-11
Test R.	3.8	7.3	0.44	1.4	8.5	0.09	0.0	2.1	0.53
Profiler	0.0	5.5	0.07	0.0	8.5	0.02	0.0	10.4	0.01

An unexpectedly large fraction of performance bugs (50.9%, 33.9%, and 57.3% for JDT, SWT, and Mozilla, respectively) are found through code reasoning. For example, the report for bug 108820 in JDT states: *"When computing a hierarchy on a class, we should ignore potential subtypes in the index that are interfaces and annotations as these cannot possibly extend the focus class."*. This text suggests that the bug reporter understands the high level semantics of the code, knows that some computation is unnecessary, and proposes to skip that computation. From the bug report and the subsequent discussion, the reporter did not experience a slow program behavior (which would qualify the bug for Observe category), nor did the reporter profile the code to find this deficiency (which would qualify the bug for Profiler category).

In contrast, only few non-performance bugs (5.1%, 4.1%, and 15.5% for JDT, SWT, and Mozilla, respectively) are found through code reasoning. These differences are statistically significant as the p-values in row *Reason* are less than 0.05.

Contrary to expectations, the table shows that profiling code is not the major source for discovering performance bugs, accounting for only 5.5%, 8.5%, and 10.4% of performance bugs in JDT, SWT, and Mozilla, respectively.

Some reporters found bugs using a mixture of the above techniques, and we label such reports in the group with the most precise technique, or the technique that seemed to be the primary factor that contributed to finding that bug. For example, the report for bug 79557 in SWT states: *"Display.getShells() and Disply.getActiveShell() methods are called in eclipse very frequently. ... This is due to widgetTable, which contains hundreds of widgets, is scanned each time. However amount of non-disposed shells is about 2-4 depending on amount of opened dialogs. So it is better to keep separate array of non-disposed shells rather than scan throw [sic] widgetTable."*. The developers likely profiled an execution scenario, identified some method as being expensive, and then reasoned about the code and tried to deduce if that method usage pattern can be improved. We consider the report in the Profiler category, because the primary means to find the bug was profiling, not purely code reasoning.

The above data suggests that developers need tool support to detect performance bugs. For example, static analysis may help developers during code reasoning and better profilers may focus on finding performance bugs rather than only slow computation (which may be truly needed and thus not a bug).

Unlike non-performance bugs, many performance bugs are found through code reasoning, not through direct observation of the bug's negative effects. Few performance bugs are found through profiling.

Table 2.10 shows what additional information was provided with the bug reports for performance and non-performance bugs. Row *Input* gives the percentage of bug reports that contain a test or input file, row *Steps* gives the percentage of bug reports that have a detailed description on how to reproduce the bug but do not contain a test or input file, and row *Gen./None* is the percentage of the bug reports that contain only a high level description of the bug cause or none at all.

What additional information is provided with the bug report is not necessarily dependent on how the bugs were discovered (Table 2.9). For example, even if JDT bug 108820 was discovered through code reasoning, and thus initially there was no code exposing the bug, the developer still

Table 2.10: What additional information was provided with the bug report? This table uses the I-Perf and I-NonPerf datasets.

How?	JDT (%)		SWT (%)		Mozilla (%)	
	NPerf	Perf	NPerf	Perf	NPerf	Perf
Input	58.2	16.4	45.2	39.0	17.2	20.8
Steps	15.2	10.9	21.9	6.8	24.1	17.7
Gen./None	26.6	72.7	32.9	54.2	58.6	61.5

provides a test exposing the bug. The test was written to measure the performance of the buggy method `newTypeHierarchy(null)`, and the test does not represent a real-world usage scenario. This is similar to how developers can find non-performance bugs in some actual usage scenario but provide small unit tests that expose the bug independently of the original usage scenario that exposed the bug.

For JDT and SWT, non-performance bugs are more likely to be reported with an input or steps to reproduce than performance bugs. For Mozilla, the ratio of bugs in the Input or Steps category is about the same for performance and non-performance bugs, but performance bugs are more likely to be reported with a test or input than non-performance bugs. A large fraction of both performance and non-performance bug reports either contain only a high level description of the bug cause or no description at all. The percentages vary from project to project, which suggests that programming language and project reporting policy may have influenced the quality of the bug reports.

Overall, better reporting policies are needed for both performance and non-performance bugs. Tool support for capturing the relevant execution scenario, extracting unit tests from system tests, or deterministic replay can also help.

Many performance bugs are reported without inputs or steps to reproduce. Non-performance bugs have a similar problem, though to a lesser extent in JDT and SWT.

2.3 Threats to Validity

Internal Threats: We use keyword search and manual inspection to identify the performance bugs in I-Perf. The precision of this approach is 100%, which is the proportion of true performance bugs among the performance bugs manually verified by us. To minimize the risk of incorrect results given by manual inspection, the bugs in I-Perf were labeled as performance bugs independently by two authors. We estimate the recall of this technique to be 50%, which means that for each performance bug that we analyzed, there is a performance bug that we missed. To compute this recall, we randomly sampled 227 bugs and manually inspected each of them. We found 6 performance bugs, of which only 3 were found by keyword search and manual inspection. Zaman et al. [92] use an alternate approach to compute recall. They sample approximately the same number of predicted performance and non-performance bugs. Using this approach, our recall is 97.22%, which is comparable to the recall obtained by them. The risk of not analyzing all performance bugs cannot be fully eliminated. However, combining keyword search and manual inspection is an effective technique to identify bugs of a specific type from a large pool of generic bugs, which was successfully used in prior studies [27, 34, 48].

External Threats: The bugs we use are from relatively large and mature applications, written both in Java (JDT and SWT) and in C/C++ (Mozilla). However, we cannot guarantee that our results from them will generalize to all other software projects. Furthermore, the applications used in our study are open-source, and performance bugs in commercial software may have different characteristics. As in the prior study [68], the studied period is about two years, which could be a threat. Extending this study to other projects and longer periods of time remains as our future work. Data recorded in bug tracking systems and code version histories can have a systematic bias relative to the full population of bug fixes [9] and can be incomplete or incorrect [6]. Our study, like similar studies, can be affected by these problems, and minimizing the their effects is an ongoing research problem.

Construct Threats: The bugs we use were identified by Park et al. [68] by automatically

finding bug IDs from commit logs and bug data bases. While this technique can miss bugs and patches, there is no reason to believe that there are fundamental differences in the characteristics of the missed bugs and patches. The studied bugs may have not yet been fully fixed, or may be re-opened in future. To minimize this concern, the studied bugs were reported between 2003 and 2006 (Table 2.1), and thus chances are high they are fully fixed and will not be re-opened. I-Perf and I-NonPerf contain equal numbers of bugs, which does not model the fact that there are more non-performance than performance bugs. While sampling proportionally more non-performance bugs would closer model the bug population, the manual effort would be extremely high. We believe the large number of manually inspected bugs (210 performance and 210 non-performance bugs) reduces the potential risks created by this design choice.

Conclusion Threats: For the experiments where we randomly sampled bugs (e.g., I-NonPerf are a small fraction of non-performance bugs sampled out of all 14,050 non-performance bugs), the number of random samples may not be sufficient to accurately characterize the bug population. To minimize this threat, we manually inspected a large number of bugs: 210 performance and 210 non-performance bugs.

2.4 Summary

Performance bugs create problems even for well tested software written by expert programmers [39, 52, 54, 56, 70]. This chapter studies three large, mature, and popular projects (Eclipse JDT, Eclipse SWT, and Mozilla), which reveals several important findings. First, fixing performance bugs is about as likely to introduce new functional bugs as fixing non-performance bugs. Developers do not need to be overly concerned that fixing performance bugs carries a greater risk of introducing new functional bugs than fixing typical, non-performance bugs. Second, we find that fixing performance bugs is more difficult than fixing non-performance bugs. Finally, unlike non-performance bugs, many performance bugs are found through code reasoning, not through direct observation of the bug's negative effects (e.g., slow code). Furthermore, few performance bugs are found through

profiling. The results suggest that improving techniques for discovering, reporting, and fixing performance bugs would greatly help developers address performance bugs.

Chapter 3

TODDLER: Detecting Performance Bugs via Similar Memory-Access Patterns

This chapter presents the contributions of the TODDLER technique, which provides an automated oracle for detecting performance bugs. This chapter is organized as follows. Section 3.1 presents our running example, Section 3.2 presents the design and implementation details of TODDLER, Section 3.3 presents our evaluation of TODDLER, Section 3.4 discusses other design choices for TODDLER, and Section 3.5 summarizes this chapter.

3.1 Study of Performance Bugs

We study over 100 performance bugs from open-source projects to identify how these bugs depend on loops. We study both Java and C/C++ projects to obtain more generality of our findings. These bugs were collected independently of TODDLER in a recent study on performance bugs [34], but their relationship to loops was not analyzed in detail.

Our study shows that about 90% of performance bugs involve loops, and more than 50% of performance bugs involve at least two levels of loops. The bugs that involve nested loops can be categorized along two dimensions:

- Is the performance problem in the inner or the outer loop?
- Is the performance problem caused by redundant computation or inefficient computation? We define *redundant computation* as the same computation being unnecessarily repeated on the same set of data with the same result.

We next describe the four types of real-world performance bugs categorized along the above

```
1  // Simplified from the XYPlot class in JFreeChart
2  public void render(...) {
3    for (int item = 0; item < itemCount; item++) { // Outer Loop
4      renderer.drawItem(...item...); // Calls drawVerticalItem
5    }
6  }
7  // Simplified from the CandlestickRenderer class in JFreeChart
8  public void drawVerticalItem(...) {
9    int maxVolume = 1;
10   for (int i = 0; i < maxCount; i++) { // Inner Loop
11     int thisVolume = highLowData.getVolumeValue(series, i).intValue();
12     if (thisVolume > maxVolume) {
13       maxVolume = thisVolume;
14     }
15   }
16   ... = maxVolume;
17 }
```

Figure 3.1: A JFreeChart bug with a redundancy in the outer loop

two dimensions, and then discuss how this understanding of real-world bugs can guide our bug-detection design. For space reasons, we will give code examples only for two categories (but covering both inner and outer loops, as well as redundant and inefficient computation).

3.1.1 Categories of Severe Performance Bugs

Category 1 (Redundancy in Outer Loops): Redundant computation is conducted across iterations of an outer loop. This redundant computation involves an expensive inner loop, which makes the performance problem severe. Problems of this type are usually difficult for compilers to optimize, because they involve nested loops and usually many functions. They are usually fixed by storing and reusing results from previous loop iterations.

Figure 3.1 demonstrates such a bug from JFreeChart, a popular Java framework for drawing charts. This bug is particularly severe, because it causes the chart display to freeze. The outer loop iterates over all the items in a data set (line 3) and for each item calls the method `drawItem`, which in turns calls the method `drawVerticalItem`. The inner loop (line 10) in `drawVerticalItem` computes the maximum volume (line 12) of all the items in the data set. The repeated computation of maximum is redundant, because the volumes of the items do not change between calls. Thus, the inner loop can be performed only once, not in every iteration of the outer loop. Indeed, to fix this bug, the developer changed the code to cache and reuse the maximum volume.

```
1  // SetDecorator class in Google Core Libraries contained this method call
2  set.removeAll(arrayList);
3  // Simplified from the AbstractSet class in the standard Java library
4  public boolean removeAll(Collection<?> c) {
5    if (size() > c.size()) {
6      for (Iterator<?> i = c.iterator(); i.hasNext(); )
7        remove(i.next());
8    } else {
9      for (Iterator<?> i = iterator(); i.hasNext(); ) { // Outer Loop
10       if (c.contains(i.next())) {
11         i.remove();
12       }
13     }
14   }
15 }
16 // Simplified from the ArrayList class in the standard Java library
17 public boolean contains(Object o) {
18   for (int i = 0; i < size; i++) { // Inner Loop
19     if (o.equals(elementData[i]))
20       return true;
21   }
22   return false;
23 }
```

Figure 3.2: A Google Core Libraries bug with an inefficient inner loop. This was a *previously unknown* bug found by TODDLER.

Category 2 (Redundancy in Inner Loops): Redundant computation is conducted across iterations of an inner loop. This computation waste is amplified by outer loops that dynamically call the inner loop many times. Performance problems of this type are difficult for compilers to optimize when the redundant computation involves function calls. They are usually fixed by hoisting computation out of the loop.

Category 3 (Inefficient Outer Loops): The program has an expensive but necessary inner loop. Unfortunately, this loop is inefficiently used by an outer loop, which leads to severe performance problems. Problems of this type cannot be optimized by compilers, because they require deep understanding of code. They are usually fixed by changing outer loops so that the inner loop will execute less frequently.

Category 4 (Inefficient Inner Loops): The inner loop conducts an inefficient, but not redundant, computation. This inefficiency is amplified by an outer loop that uses each iteration to execute the inner loop on a slightly different data set. Again, problems of this type cannot be optimized by compilers, because they require deep understanding of code. Their patches need to find a more efficient or incremental algorithm to replace the inner loop, which often can be achieved with a more appropriate data structure for the data set under operation.

Figure 3.2 demonstrates an example from Google Core Libraries (GCL). This is a *previously unknown bug found by* TODDLER. After we reported it, GCL developers not only fixed this bug but also searched through their entire codebase for similar code patterns and fixed 8 other classes affected by similar bugs. (We count these 9 instances as *one bug* not 9 bugs.) The GCL code called the `removeAll` method on a `Set` object, passing it an `ArrayList` object as a parameter. The `removeAll` method removes from the set `this` all the elements contained in the specified collection `c`. The method has a performance optimization that chooses whether to iterate over the set `this` or the collection `c` based on their sizes (line 5), under the assumption that the cost of `contains` and `remove` operations are similar for the set and the collection when they have similar sizes. In the `else` branch, the outer loop iterates over each element of `this` and checks if `c` contains the element (lines 9–13).

When `c` is an `ArrayList`, `contains` performs a linear search (lines 18–21), which is inefficient, so it would have been better to iterate over `c` and call `remove` on the set because it has a more efficient inner loop. Indeed, the GCL developers changed their code, replacing the call to `removeAll` by conceptually inlining the body of `removeAll` and keeping only the `then` branch from the body. In general, the solution for this category is to simplify the inner-loop computation.

3.1.2 Implications

Why do developers need automated support for performance testing? The above examples demonstrate that many performance bugs are difficult to avoid, because they involve library functions or APIs whose performance features are opaque to developers. In addition, a lot of time-consuming computation, such as many inner loops in our examples, is embedded in code written by different developers. As shown in Figure 3.2, GCL developers did not initially consider that the performance of the Java library method `AbstractSet.removeAll` is sensitive to the data structures used for parameters, and this information is not even stated in the documentation for `removeAll`. Tool support is needed to help developers detect these hard-to-avoid performance bugs.

Why do we focus on nested-loop performance bugs? Bugs that involve nested loops usually have severe performance impact. The reason is that the inner loop represents an expensive computation inside the outer loop, and the outer loop amplifies the performance penalty of the inner loop. For example, in the JFreeChart bug from Figure 3.1, the inner loop is slow, but if executed only once, it cannot have a significant effect on performance; however, if executed many times in the outer loop, it causes the chart display to freeze.

How can we detect nested-loop performance bugs? A common feature of above nested-loop performance bugs is that they often involve *repeated memory-access patterns*. Bugs from Category 1 conduct redundant computation across outer-loop iterations. A big chunk of the computation in each outer-loop iteration repeats the computation from an earlier iteration with the same input and the same result. Hence, outer-loop iterations share long sequences of memory reads that return the same values. For example, the iterations of the outer loop in Figure 3.1 share a long sequence of reads inside the `intValue` method (line 11). Bugs from Category 2 conduct redundant computation during every iteration of an inner loop, which results in memory reads that repeatedly return the same value. Bugs from Categories 3 and 4 have less regular patterns than bugs from Categories 1 and 2, but the memory-access similarities are still strong. The outer-loop iterations in bugs from Categories 3 and 4 often work on similar data sets. That is the reason why developers can effectively optimize these bugs. That is also the reason why there are usually memory reads that return similar sequences of values across outer-loop iterations. In sum, looking for repeated memory-access patterns is an effective way to look for performance bugs from all four categories.

3.2 TODDLER Design and Implementations

Motivated by the study in Section 4.1, we have developed TODDLER, an automated oracle that finds likely performance bugs by looking for loops that read similar sequences of values across iterations. TODDLER considers such similar sequences to be a strong indication of redundant or inefficient computation and reports such loops as performance bugs.

TODDLER is a dynamic technique. It instruments the code under test, runs each test from a given test suite, and reports only the tests that contain loops with similar sequences. We first describe the instrumentation that TODDLER adds. We then describe the data structures and algorithms that TODDLER uses for storing information about reads and finding similarity among sequences. We finally discuss our two implementations of TODDLER in a full-blown tool for Java and a simple prototype for C/C++.

3.2.1 Instrumentation

To monitor loops and read instructions, TODDLER instruments the code, both the application under test and the libraries it depends on, because many performance bugs are caused by the misuse of libraries. For loops, the instrumentation is straightforward: TODDLER analyzes the code, assigns a unique ID for each static loop, and inserts in the code three types of method calls that inform the TODDLER runtime whenever a loop starts, a loop iteration starts, or a loop finishes. For read instructions, the instrumentation itself is also simple: for each instruction that reads object fields or array elements from the heap (e.g., Java bytecode instructions `GETFIELD` or `AALOAD`), TODDLER inserts a method call that informs the TODDLER runtime about the value read by the instruction and the call stack within which the instruction is executed. Note that TODDLER identifies a read instruction by both the static occurrence of the instruction in the code *and* the dynamic context (i.e., the call stack) in which the instruction executes. We use the term *IPCS* (instruction pointer + call stack) to refer to a static instruction with its dynamic context.

3.2.2 Collecting IPCS-Sequences

We use the term *IPCS-sequence* to refer to the sequence of values read by all dynamic instances of an IPCS I during an iteration of a loop L. Note that, when I is inside an inner loop of L, the IPCS-sequence for the outer loop L is likely to contain more than one element. Also note that TODDLER builds *one IPCS-sequence per IPCS* rather than one IPCS-sequence per the entire loop

```
1  StartLoop(L1)
2  StartIter    Read(i1, v1)
3               StartLoop(L2)
4               StartIter Read(i2, v2)
5               StartIter Read(i2, v3) Read(i3, v4)
6               StartIter Read(i3, v5)
7               StartIter Read(i2, v6) Read(i3, v7)
8               FinishLoop(L2)
9  StartIter
10              StartLoop(L2)
11              StartIter Read(i2, v8)
12              StartIter Read(i2, v9) Read(i3, v10)
13              StartIter Read(i2, v11) Read(i3, v12)
14              StartIter Read(i3, v13)
15              FinishLoop(L2)
16              Read(i4, v14) Read(i5, v15)
17 FinishLoop(L1)
```

Figure 3.3: Example events produced by running instrumented code

iteration, and thus a loop iteration has as many IPCS-sequences as it has IPCSs.

To illustrate, Figure 3.3 shows an example stream of events produced when some instrumented code is executed; $\mathtt{i}_N$ represents an IPCS, and $\mathtt{v}_M$ represents a value read. From these events, TODDLER creates IPCS-sequences of values read by the same IPCS during a loop iteration. For example, for the *outer* loop L1, TODDLER would create IPCS-sequences $\mathtt{i}_1$:[$\mathtt{v}_1$], $\mathtt{i}_2$:[$\mathtt{v}_2$,$\mathtt{v}_3$,$\mathtt{v}_6$], and $\mathtt{i}_3$:[$\mathtt{v}_4$,$\mathtt{v}_5$,$\mathtt{v}_7$] for the first iteration and $\mathtt{i}_2$:[$\mathtt{v}_8$,$\mathtt{v}_9$,$\mathtt{v}_{11}$], $\mathtt{i}_3$:[$\mathtt{v}_{10}$,$\mathtt{v}_{12}$,$\mathtt{v}_{13}$], $\mathtt{i}_4$:[$\mathtt{v}_{14}$], and $\mathtt{i}_5$:[$\mathtt{v}_{15}$] for the second iteration.

Note that the IPCS-sequences for the innermost loops have length 1, e.g., for the first dynamic instance of the *inner* loop L2, the IPCS-sequences would be just $\mathtt{i}_2$:[$\mathtt{v}_2$], $\mathtt{i}_2$:[$\mathtt{v}_3$], and $\mathtt{i}_2$:[$\mathtt{v}_6$] for $\mathtt{i}_2$ and similar for $\mathtt{i}_3$. Also note that an IPCS need not occur in every iteration of a loop (e.g., $\mathtt{i}_2$ does not occur in the third iteration of the first dynamic instance of L2). In that case, TODDLER still creates an IPCS-sequence (for L1) of consecutive values read for the same IPCS even if these values are not read in consecutive loop iterations (of L2).

While this example illustrates TODDLER only on the loop nesting depth of two, TODDLER handles larger nesting depths in the same manner, by appending IPCS-sequences for the same IPCS. For example, if one iteration of some loop L0 had the events shown in Figure 3.3, then for that iteration of L0, TODDLER would create $\mathtt{i}_1$:[$\mathtt{v}_1$], $\mathtt{i}_2$:[$\mathtt{v}_2$,$\mathtt{v}_3$,$\mathtt{v}_6$,$\mathtt{v}_8$,$\mathtt{v}_9$,$\mathtt{v}_{11}$], $\mathtt{i}_3$:[$\mathtt{v}_4$,$\mathtt{v}_5$,$\mathtt{v}_7$,$\mathtt{v}_{10}$,$\mathtt{v}_{12}$,$\mathtt{v}_{13}$], $\mathtt{i}_4$:[$\mathtt{v}_{14}$], and $\mathtt{i}_5$:[$\mathtt{v}_{15}$].

```
1  // Instruction pointer and its dynamic context
2  class IPCS { int IP; CallStackHash cs; }
3  // Value of a memory location
4  class Val { long val; }
5  // IPCS-sequence of values read by an IPCS in one iteration
6  class Seq { List⟨Val⟩ list; }
7  // Dynamic loop record
8  class DynLoop {
9    int id; // static id of the loop
10   CallStackHash cs; // calling context
11   int iterations; // number of iterations
12   // map each IPCS encountered during loop execution...
13   // ...to values read by the IPCS in the iterations
14   Map⟨IPCS , List⟨Seq⟩⟩ map;
15 }
```

Figure 3.4: Data structures for storing and processing IPCS-sequences

3.2.3 Data Structures

Figure 3.4 shows the data structures that TODDLER uses to store information about loops. `IPCS` has an `IP` that statically determines the instruction (e.g., its class, method, and bytecode offset within the method in Java) and the call stack that represents the dynamic context in which the instruction executes. (Call stacks can be efficiently computed using hashing [10].) `Val` represents a value read by an instruction, which is either a primitive value or an object ID (obtained with `System.identityHashCode()` in Java). Note that the ID is of the object being *returned* by the read, not of the object being *dereferenced*. For example, in `e.next`, the ID is of `e.next` not of `e`. `Seq` is an IPCS-sequence of values read by the same `IPCS` in one loop iteration. `DynLoop` records information about one dynamic loop instance: the static loop ID (its class, method, and bytecode offset within the method in Java), the call stack in which the loop executes, the number of loop iterations, and the IPCS-sequences across all iterations for each `IPCS`. For example, for i_2, the two IPCS-sequences of the *outer* (L1) loop are i_2:[[v_2,v_3,v_6],[v_8,v_9,v_{11}]].

3.2.4 Algorithm for Finding Performance Bugs

Figure 3.5 shows the pseudo-code of the top-level function. TODDLER checks for potential performance bugs in each dynamic loop that had more than a few iterations (by default, `minIter`=10; this threshold is a configurable parameter of our algorithm, and Section 3.3.4 discusses the impact of the parameters). For each `DynLoop`, TODDLER finds all `IPCS`s that have similar IPCS-sequences

```
1  // One parameter for loops
2  int minIter; // absolute number of loop iterations
3
4  // Input: the record of a dynamic loop
5  // Output: whether this loop has performance bugs
6  boolean hasPerformanceBug(DynLoop loop) {
7    return !(computeSimilarIPCSs(loop).empty());
8  }
9
10 // Input: the record of a dynamic loop
11 // Output: IPCSs that read similar values across iterations
12 Set<IPCS> computeSimilarIPCSs(DynLoop loop) {
13   Set<IPCS> similarIPCSs = new Set<IPCS>();
14   // ignore very small loops
15   if (loop.iterations < minIter) return similarIPCSs;
16   for (curIPCS : loop.map.keyset())
17     // compare IPCS-sequences for iterations in which curIPCS occurs
18     if (areSimilarIterations(loop.map.get(curIPCS)), loop.iterations)
19       similarIPCSs.add(curIPCS);
20   return similarIPCSs;
21 }
```

Figure 3.5: The top-level function for TODDLER

across loop iterations. If there is any such `IPCS`, TODDLER reports a performance bug.

Given a test suite, TODDLER runs each test, collects `DynLoop` objects, and reports a set of static loops that have similar IPCS-sequences for at least one test. For each static loop, TODDLER generates a set of records that help in understanding and debugging the problem. Each record contains the test that executes the loop, the call stack for the loop, the static IP of the instruction that reads similar values, the call stack for that instruction, and statistics about similarity.

Note that TODDLER can find the same loop to be repetitive for multiple tests. Rather than printing a report for each test and each loop, TODDLER *clusters* these reports based on the *static* outer loop. Clustering is commonly used for grouping failure reports in testing [69, 89].

3.2.5 Measuring Similarity

Figure 3.6 shows the pseudo-code for finding similar IPCS-sequences across loop iterations. TODDLER compares consecutive IPCS-sequences for the same `IPCS`. As mentioned in Section 3.2.1, an IPCS may not be executed in every iteration of a loop. TODDLER computes the ratio of the number of IPCS-sequences to the number of loop iterations and ignores `IPCS`s that occur in a small ratio of iterations, because even if the computation at these `IPCS`s is similar and could be optimized, they may not be an expensive part of the entire loop. (By default, `minSeqRatio`=45%.)

```
1  // Two parameters for loop iterations
2  float minSeqRatio; // relative number of IPCS-sequences in the loop
3  float minSimRatio; // relative number of similar iterations
4
5  // Input: IPCS-sequences for all iterations of a loop
6  // Output: whether IPCS reads similar values across iterations
7  boolean areSimilarIterations(List⟨Seq⟩ seqs, int iterations) {
8    // ignore IPCS that occurs in a small fraction of iterations
9    if ((seqs.size() / iterations) < minSeqRatio) return false;
10   int similar = 0;
11   for (int i = 0; i < seqs.size()−1; i++)
12     if (areSimilarSequences(seqs[i], seqs[i+1])) similar++;
13   return (similar / (seqs.size()−1)) >= minSimRatio;
14 }
```

Figure 3.6: Checking the similarity throughout a loop

To compare the IPCS-sequences of an IPCS inside a loop L, TODDLER determines whether these IPCS-sequences are similar *throughout L* based on the relative number of similar *consecutive* IPCS-sequences. The IPCS-sequences are considered similar throughout loop L if and only if the ratio is larger than the threshold. (By default, `minSimRatio`=70%.)

Redundant and inefficient computation can be reflected not only by IPCS-sequences that are exactly the same across iterations, such as the IPCS-sequences from `intValue()` in Figure 3.1, but also by IPCS-sequences that are slightly different across iterations, such as the IPCS-sequences for `elementData[i]` in Figure 3.2. Thus, we need to judge whether two IPCS-sequences are *similar* enough to represent potential performance problems.

Figure 3.7 shows the pseudo code of this algorithm. TODDLER uses the *longest common substring* [19] to measure the similarity between two IPCS-sequences. (Note that *substring* refers to the *consecutive* occurrences of values in the IPCS-sequences, while *subsequence* would refer to the *potentially non-consecutive* occurrences of values.) The longest common substring can be computed in $O(nm)$ time where n and m are the lengths of the two IPCS-sequences [19]. We define two IPCS-sequences to be similar if both the absolute and relative length of their longest common substring are above thresholds. (By default, `minLCS`=7 and `minLCSRatio`=70%.)

```
1  // Two parameters for IPCS-sequences of values
2  int minLCS; // absolute length of the longest common substring
3  float minLCSRatio; // relative length of the longest common substring
4
5  // Input: two IPCS-sequences
6  // Output: whether two IPCS-sequences are similar
7  boolean areSimilarSequences(Seq S1, Seq S2) {
8    lcs = longestCommonSubstring(S1, S2).size();
9    lcsRatio = lcs / min(S1.size(), S2.size());
10   return (lcs >= minLCS) && (lcsRatio >= minLCSRatio);
11 }
```

Figure 3.7: Checking the similarity of two IPCS-sequences

3.2.6 Filtering Reads

TODDLER can filter reads that have repetitive values but are unlikely to indicate performance bugs. First, TODDLER ignores IPCS-sequences that repeat only one value. For example, an inner loop of the form `for (int i = 0; i < this.size; i++)` repeatedly reads the value for `this.size` but does not contain a performance bug. Note that this heuristic may cause TODDLER to lose some Category 2 bugs. For example, if `this.size` is returned by a synchronized getter method, which is slower than just reading `this.size`, one may want to pull the getter method call out of the loop. TODDLER considers all operations to take an equal amount of time, and therefore does not report the repeated getter method calls as a performance bug. Future implementations can add timing information to TODDLER.

Second, TODDLER for Java ignores reads that happen in the class initializer methods because these are executed only once per class loading, so even if the code contains a bug, developers may not want to change it. Third, TODDLER allows the users to specify a set of fields and methods to be ignored, when the users do not expect them to be indicative of performance bugs. TODDLER ignores IPCSs that either read a specified field or execute in a context where a specified method is on the call stack. For example, some fields are used as *indexes* and can appear in an inner loop as `for (...) { ... this.cursor++; ... }`; if the outer loop resets `cursor`, the IPCS-sequence would repeat, but repeatedly reading the index itself does not indicate inefficient or redundant computation. As another example, appending strings in a loop often leads to repeated work, and in fact, it is an anti-pattern in Java to append many `String` objects. However, to simplify coding, many

times developers do append strings in loops, and may not want to be bothered with reports of such coding patterns. By default, TODDLER ignores *only three fields and four* `toString`/`append` *methods* from the standard JDK library `java.util` classes. Note that specifying these library fields and methods is done *only once* for all applications that use the library.

3.2.7 Implementations

We implemented the TODDLER technique in a full-blown tool for Java, which we also call TODDLER, and a simple prototype for C/C++. Our Java implementation is based on static Java-bytecode instrumentation, using Soot 2.4.0 [77]. TODDLER uses Soot to instrument every instruction that reads an object field or an array element, the start of each loop, the start of each loop iteration, and the exit of each loop. The implementation closely follows the pseudo-code algorithms presented earlier. It performs similarity checks *online*, i.e., collects IPCS-sequences of values read in a `DynLoop` object and, whenever the program exits a loop, calls the `hasPerformanceBug` function from Figure 3.5 to process the `DynLoop` object and decide if there is a performance bug. Section 3.3.5 discusses our C/C++ prototype.

3.3 Experimental Results

Our evaluation focuses on the Java version of TODDLER and uses 9 popular Java codebases. Figure 3.8 lists basic information about these codebases. We first evaluated TODDLER on 11 previously known real-world performance bugs and on over 173,000 existing functional tests from these codebases. We then *settled on the values for the* TODDLER *parameters* and evaluated it on newly written performance tests. Our experiments found *42 real-world performance bugs* in these codebases (39 in the application code and 3 in the libraries they use).

The rest of this section first presents our experiments with the 11 previously known bugs. It then presents our experiments with performance tests and the new bugs that we found. It next presents the evaluation with the existing functional tests. It finally presents a sensitivity analysis

ID	Application	Description	LoC	Known Bugs	New Bugs
#1	Ant	build tool	109,765	1	8
#2	Apache Collections	collections library	51,416	1	20
#3	Groovy	dynamic language	136,994	1	0
#4	Google Core Libraries	collections library	156,004	2	10
#5	JFreeChart	chart framework	64,184	1	1
#6	JMeter	load testing tool	86,549	1	0
#7	Lucene	text search engine	320,899	2	0
#8	PDFBox	PDF framework	78,578	1	0
#9	Solr	search server	373,138	1	0
	JDK standard library				2
	JUnit testing framework				1
SUM				**11**	**42**

Figure 3.8: Applications used in experiments, previously known bugs, and new bugs found with TODDLER.

of the parameter values. Unless otherwise specified, all the experiments use the following default values: `minIter`=10, `minSeqRatio`=45%, `minSimRatio`=70%, `minLCS`=7, `minLCSRatio`=70%.

We conduct all experiments where time is measured on an AMD Athlon machine with 2.1GHz CPU, 3GB memory, and Oracle/Sun JVM version 1.6.0. We also conduct experiments where time is not measured on a cluster of machines; while TODDLER does not need a cluster for regular use, we needed it for our extensive experiments.

3.3.1 Experiments with Previously Known Bugs

To evaluate bug-detection coverage, accuracy, and overhead of TODDLER, we first used 11 known real-world bugs from the 9 codebases. We searched the respective bug-tracking databases to collect these bugs; they were reported by the users of these applications and the bug description clearly marks them as performance bugs.

We run TODDLER on a performance test related to the bug report for each of the 11 bugs. Because each test is supposed to reveal a bug, we effectively evaluate if TODDLER has *false negatives* that miss some bugs. We compare the results of TODDLER with the results of a traditional profiler ran on the same tests. As explained in Section 1.3, profilers are not designed to detect performance

Known Bug	Bug Detected?		False P.	Rank	Slowdown	
	TODD.	HPROF	TODD.	HPROF	TODD.	HPROF
#1	✓	-	0	19.3	13.7	4.2
#2	✓	✓	0	1.0	10.0	2.1
#3	✓	✓	0	3.7	15.5	3.7
#4.1	✓	✓	0	1.8	9.0	3.8
#4.2	✓	-	0	5.3	7.5	3.2
#5	✓	-	0	53.7	13.4	8.8
#6	✓	-	0	10.3	8.5	1.9
#7.1	✓	-	0	7.7	6.8	2.5
#7.2	✓	✓	0	3.1	25.4	3.1
#8	✓	-	1	18.8	51.8	12.1
#9	✓	-	0	178.3	114.2	7.1
SUM	**11**	**4**	**1**		**15.9X**	**4.0X**

Figure 3.9: Comparison of TODDLER and HPROF for bug-triggering tests.

bugs, but are the only traditional tool that developers could use without TODDLER. Specifically, we use HPROF [67], the standard Java profiler. It outputs a ranked list of methods (more precisely, calling contexts) that consume the most time. We measure how highly HPROF ranks the buggy method (that contains the buggy code region). Additionally, for these 11 tests, we compare the run-time overheads of TODDLER and HPROF.

Bug Detection Results

Figure 3.9 summarizes the results for the 11 bugs. TODDLER finds all the bugs (no false negatives) and produces only one false positive. Specifically, for bug #8, TODDLER produces two reports: one showing the real bug and one being a false positive. (Section 3.3.3 discusses false positives.) TODDLER finds these 11 bugs because they involve at least two levels of loops and have similar sequences of values read across loop iterations. In fact, most of these bugs have so strongly similar sequences that TODDLER can detect them under a wide range of threshold settings. (Section 3.3.4 discusses sensitivity to threshold settings.)

Figure 3.9 also shows the results for HPROF. We use it with the `cpu=times` option as it gives more accurate results than `cpu=samples`, though at a higher overhead. However, even with `cpu=times`, the results of HPROF for the same code and same input can vary from one run to

another. Therefore, we ran each test under HPROF 10 times and show the mean ranking of the buggy method.

The developer is unlikely to inspect more than a handful of methods reported by a profiler. If we consider that HPROF correctly detects a bug when the buggy method ranks in top 5, then HPROF detects only 4 out of 11 cases that TODDLER detects. On the positive, HPROF ranks bug #2 consistently as number one. On the negative, for 5 out of 11 bugs, HPROF does not rank the buggy method even in the top ten. For example, bug #9 comes from a text-search server, Solr. The method with the performance bug constructs a set of strings that represent filter keywords. Under normal server setting, this set is small, and the method consumes only about 0.1% of the total search-query time. As a consequence, it ranks only about 178th in the profiling results.

A careful reader may wonder if an easier approach would suffice to find the bugs that TODDLER finds: could we simply report all nested loops as potentially buggy? We added code to count nested loops during an execution, more precisely static outer loops that dynamically execute at least one inner loop. For the 11 tests, the number of such outer loops ranges from 1 to 12, and the total number of such loops is 38. Thus, a naïve technique that reports every nested loop as a performance bug would have 27(=38-11) false positives for just these 11 bugs. In contrast, TODDLER can identify truly performance-wasting nested loops by analyzing memory-access patterns and reports only one false positive for these 11 cases.

Performance Results

The last two columns of Figure 3.9 show the slowdown that TODDLER and HPROF have over an execution with no tool for the 11 bug-triggering tests. TODDLER causes, on average, a 15.9X slowdown that comes from monitoring read accesses and comparing IPCS-sequences. Our current implementation of TODDLER is about 4 times slower than HPROF. In the future we plan to further reduce the overhead of TODDLER through sampling techniques and static analysis.

3.3.2 Experiments with New Bugs and Performance Tests

We further evaluate bug-detection coverage and accuracy of TODDLER by applying it on *performance tests*, which is the intended usage scenario for TODDLER. To avoid the bias of us as tool authors manually writing tests, we use three sets of tests not written by us: (1) automatically generated tests, (2) tests manually written by an undergraduate student *familiar* with performance testing ("expert"), and (3) tests manually written in a controlled experiment by 8 graduate and undergraduate students *unfamiliar* with performance testing ("novices"). We use these different sets to assess how TODDLER works for tests with various characteristics.

We focus our efforts on collection classes because they are widely used and make both automated generation [74] and manual writing of tests easier than domain-specific applications such as Groovy or Lucene. Ant, Apache Collections, and Google Core Libraries (GCL) implement collection classes. The performance tests for collections follow a simple pattern: create some empty collection(s), insert several elements into the collection(s), and finally call a method under test. (Note that performance tests need not necessarily check the functional results of the methods.) The collections for performance tests should not be very small, e.g., when testing `Collection.removeAll(Collection c)`, both `this` and `c` should have a reasonable number of elements, say, over 20 each; if they had a very small number, say, 2 each, it is unlikely the test would be useful for performance testing.

We wrote a simple library to automate generation of performance tests for collections. Our library can generate individual collections of various types, sizes, element types, and element values, e.g., generate an `ArrayList<Integer>` with elements `1-50`. Moreover, our library can generate multiple collections with various relationships in terms of types (collections of same or different types), sizes (collections of same, smaller, larger sizes), and intersection of elements (collections that are disjoint, equal, or partially intersect), e.g., generate a set with elements `1-50` and a list with elements `1-75`. Our library supports exhaustive and random selection of combinations of these relationships. The design goal for the library was not to extensively cover all the cases but to provide

Who	App	Tests	#Dyn. Loops	Bugs	Bugs in Test	False Pos.	Sum
Auto	#1	691	13,748	5	0	1	6
	#2	3,375	342,821	18	1	2	21
	#4	1,703	423,406	9	0	0	9
Ex-pert	#2	60	6,761	10	0	1	11
	#4	60	6,319	2	0	0	2
Nov-ice	#2	14	2,057	1	6	0	7
	#2	20	3,043	2	0	0	2
	#2	5	1,868	1	0	0	1
	#2	18	3,269	1	0	0	1
	#2	5	606	0	0	0	0
	#2	28	4,502	2	0	0	2
	#2	30	3,810	1	0	0	1
	#2	5	1,996	1	0	0	1
Unique Bugs Found:				**35**	**FPs:**	**4**	

Figure 3.10: Experiments with performance tests. Note that the same bug may be found by different automatically generated and manually written tests.

some reasonable tests for TODDLER.

We collected two types of manually written tests. We asked the "expert" to write tests for any methods in GCL and Apache Collections. We asked each "novice" to spend an hour writing tests for a given set of 10 methods in a class from Apache Collections; one of these 10 methods contained a known performance bug, and we wanted to check if the students would write tests that find this bug.

Figure 3.10 shows the number of tests generated/written for each codebase, the number of dynamic loops executed, and the number of reports that TODDLER produces. We examined all these reports to identify if they are real bugs or false positives.

We found 35 new, previously unknown performance bugs in Ant, Apache Collections, GCL, and even in a JDK class called from these projects; based on our reports, developers so far have fixed 8 of these bugs and confirmed 6 more as real bugs. TODDLER was highly effective in finding performance bugs using both automatically generated and manually written tests. Both types of tests found bugs, and sometimes found the same bugs. (Our study used older versions of GCL and Apache Collections, without the fixes for the bugs we reported.) Surprisingly, some "novice"-

written tests found two bugs in a class that we expected to have only one bug.

We also found 7 performance bugs where the test code itself is unnecessarily slow. For example, the "novice"-written tests had assertions that check the method results, and the assertions themselves use rather slow code, e.g., nested loops that search in lists but could have searched in sets. If such loops appeared in the code under test, they would be definite bugs that should be changed.

3.3.3 Experiments with Functional Unit Tests

The first two sets of experiments used tests written for performance, which is the intended usage scenario for TODDLER. To further evaluate TODDLER, we run it on the *functional* JUnit tests that come with the 9 codebases used in our evaluation. Note that this is *not the intended usage scenario*: a developer would *not use functional tests for performance testing* and thus would not use TODDLER on the functional JUnit tests. We perform these experiments *only* to stress-evaluate TODDLER.

Our experiments use 173,439 tests shown in Figure 3.11. These tests execute 24,810–3,526,496 dynamic loops (and 1,181,628–54,054,728 dynamic iterations) per codebase, a challenge for the run-time monitoring scalability. The tests also cover 21–919 unique static loops that contain nested loops per codebase, a challenge for the bug-detection accuracy.

TODDLER successfully ran for this extensive evaluation and reported 43 static loops as having similar memory accesses and thus potential performance bugs. We examined all these reports and found 7 real bugs. For JFreeChart (#5), one bug is in the JFreeChart code itself and the other in the standard JDK library. For Apache Collections (#2), one bug we reported is already fixed, and the other three bugs are similar to three bugs we previously reported and developers resolved by changing the Javadoc documentation to clarify the performance problems. For Ant (#1), all three bugs have been already fixed in the latest release. (Our experiments use older versions of the codebases.) For Apache Collections (#2) we also found 7 performance bugs in tests, where the test code is unnecessarily slow and would need to be fixed had it been in the application code.

App	# Tests	# Dynamic Loops	Bugs	Bugs in Test	False Pos.	Sum
#1	675	877,362	3	0	3	6
#2	31,105	3,526,496	4	7	1	12
#3	464	281,596	0	0	0	0
#4	138,997	2,574,756	0	0	4	4
#5	332	514,824	2	0	1	3
#6	164	88,548	0	0	1	1
#7	675	1,488,977	0	0	4	4
#8	42	24,810	0	0	0	0
#9	985	1,395,494	0	0	13	13
Unique Bugs Found:			**7**	**FPs:**	**27**	

Figure 3.11: Experiments on JUnit *functional* tests. Note that this is *not the intended usage scenario for* TODDLER; a developer would *not use functional tests for performance testing*.

The remaining 27 reports are false positives due to three causes. First, in 10 reports, the test input itself contains a lot of repetition and similar values, so TODDLER detects similarity due to the specific input provided, not because the computation is repetitive in general. Such false positives could be eliminated by using less repetitive test inputs. Second, in 3 reports, the code performs some computation on all possible pairs of values from two data sets. Such code is naturally repetitive, but the repetitions are useful computation, not performance bugs. Such false positives may be eliminated by analyzing the data flow of computation results, but such an analysis is beyond the scope of this chapter. Third, in 14 reports, the computation is truly repetitive, but removing the repetition would be too complex or would not provide clear speedup, so a developer is unlikely to change the code.

3.3.4 Parameter Sensitivity

The false-positive and false-negative rates of TODDLER are affected by the values for the five parameters described in Section 4.2. All these parameters provide the minimum threshold that loops/iterations/sequences need to satisfy to be deemed indicative of performance bugs. Hence, larger thresholds could lead to fewer false positives but more false negatives, while smaller thresholds could lead to more false positives but fewer false negatives. We experimented with various

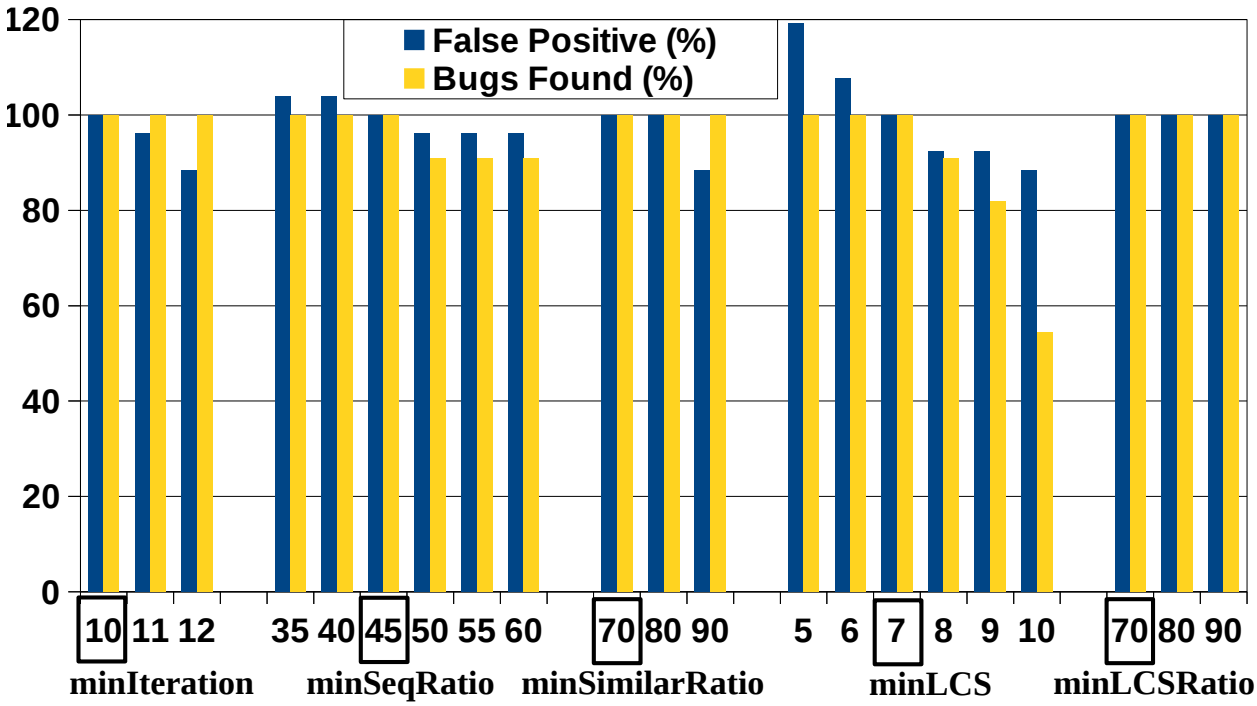

Figure 3.12: Parameter-sensitivity experiments. Each configuration changes only one threshold, with its value shown on the X-axis. The default values are boxed. Two sets of experiments are conducted for each configuration: the left/dark bar shows false positives on JUnit tests, and the right/light bar shows bugs found on bug-triggering inputs. The Y-axis shows the numbers normalized to the results under default setting.
threshold values to understand their impact.

Figure 3.12 shows the results for several configurations. For each configuration, we change only one threshold value and keep the other four at the default setting. To evaluate the impact on false negatives, we apply TODDLER on the 11 bug-triggering tests for previously known bugs (Section 3.3.1) and count the number of bugs found. To evaluate the impact on false positives, we cannot use TODDLER in the intended scenarios from sections 3.3.1 and 3.3.2, because they have few false positives. We thus use the functional tests (Section 3.3.3). The default configuration finds all 11 known bugs in the experiments from Section 3.3.1 and reports 27 false positives in the experiments from Section 3.3.3. Figure 3.12 plots the number of bugs found (light/yellow bars) and false positives (dark/blue bars) normalized to the values for the default configuration. For bugs found, higher is better, and for false positives, lower is better.

Impact on False Negatives: We increased the threshold value for each parameter, and for

only two of them such increase has caused false negatives. The most sensitive is `minLCS`, which measures the absolute length of the longest common substring between two consecutive IPCS-sequences for an instruction. When `minLCS` increases from the default 7 to 10, the number of bugs found steadily decreases from 11 to 6. The longest common substring is usually shorter than the total number of inner-loop iterations, which is often determined by the input scale. Therefore, when the input scale is small, a high `minLCS` setting could miss many bugs.

The other parameter whose increase caused false negatives is `minSeqRatio`, which measures the ratio of loop iterations that have executed the particular memory-read instruction. The inner loop of bug #7.1 is buried inside an `if` statement that is executed by about half of the outer-loop iterations. As a result, this bug is missed once `minSeqRatio` gets over 50%. We believe that this type of if/then/else situation is common enough to have the default value under 50%. Note that, except for this type of bugs, `minSeqRatio` can be increased to 60% and beyond without losing any bugs.

Impact on False Positives: For the two parameters that caused false negatives above, `minLCS` and `minSeqRatio`, we both increased and decreased the threshold values. For the other three parameters, we only increased the values. We can see that increasing `minIter` and `minSimRatio` over the default values decreases the number of false positives by about 12% *without loosing any bugs*. In practice, one may want to indeed increase these parameters, but we chose conservative parameter values. In contrast, `minLCSRatio` is the least sensitive: increasing it from 70% to 90% changes neither false positives nor false negatives.

Choosing the Threshold Values: As seen from the discussion above, TODDLER can work well in a large range of threshold values. Note that *we did not choose the default threshold values for* TODDLER *to obtain the best results for false positives and false negatives*. For example, we could increase `minIter` and `minSimRatio` to get fewer false positives without missing any bug. Rather, we chose the default values based on our intuition about the values that could give reasonable results. Moreover, we settled on these values *before* running TODDLER on performance tests (Section 3.3.2).

3.3.5 TODDLER for C/C++ Code

The performance bugs that TODDLER finds do not exist only in Java code; as already mentioned in Section 4.1, such bugs also exist in C/C++ code. To further evaluate our technique, we implemented a prototype TODDLER tool for C/C++ code. Our prototype uses Pin [49] to automatically instrument memory reads but currently does not automatically instrument loops; we manually added loop events for six real-world bugs (three from GCC, two from Mozilla, and one from MySQL). The prototype logs values read and loop events, and computes similarity *offline* by processing these logs using Python. The results show that this prototype can find all these six bugs. Because we do not instrument all the loops, we cannot measure false positives for this prototype.

3.4 Discussion

Loop Nesting: TODDLER misses bugs that are not in nested loops. We intentionally focused on nested loops, because they create more severe performance hits. However, non-nested loops can also be slow, e.g., loops that contain I/O. TODDLER can be extended to look for bugs in such loops by *modeling* the native, I/O methods in Java [79] to make their loops explicit.

Other Performance Bugs: TODDLER misses several categories of performance bugs, including (1) performance bugs specific to multi-threaded code such as lock contention [78], load imbalance [66], or false sharing [47], (2) bugs related to idle time [3], and (3) object bloat [85]. TODDLER finds performance bugs involving loops, which the existing techniques miss, so TODDLER complements these techniques.

Dynamic Technique: Just like profilers, TODDLER requires a test input. Fortunately, developers already write some performance benchmarks but typically measure only real time and look for regressions. TODDLER provides an oracle to identify performance bugs and encourages developers to write performance tests. As our evaluation shows, performance tests are relatively easy to write manually even by developers who are not familiar with performance testing, and one can sometimes even use automated test-generation techniques for performance tests. Future work can

focus on developing specialized test-generation techniques for performance bugs.

Similarity Measures: Because the longest common substring worked quite well for comparing similarity of IPCS-sequences, we did not evaluate any other approach. Future work could, for example, use edit distance to compare IPCS-sequences or, even further, capture the memory accesses not as IPCS-sequences of values but as execution trees that encode loop iterations and then measure tree similarity.

3.5 Summary

Performance testing would greatly benefit from automated oracles for performance bugs. We presented TODDLER, a novel oracle that detects performance bugs by identifying repetitive memory read sequences across loop iterations. TODDLER found 42 new bugs in 6 popular codebases: Ant, Google Core Libraries, JUnit, Apache Collections, JDK, and JFreeChart. So far developers have already fixed 10 of these bugs and confirmed 6 more as real bugs. We also evaluated TODDLER with 11 previously known, real-world performance bugs, and the experiments show TODDLER can effectively detect performance bugs with a much higher accuracy than profiling tools. TODDLER can help expose more performance bugs before software release by discovering problems even before they manifest as slow computation found by profilers. While these results are highly promising, TODDLER is just a starting point in addressing loop-related performance bugs.

Chapter 4

LULLABY: Detecting and Fixing Performance Bugs that Have Non-intrusive Fixes

This chapter presents the contributions of the LULLABY technique, which is a technique for automatically detecting and fixing performance bugs that have non-intrusive fixes. This chapter is organized as follows. Section 4.1 gives examples of bugs found by LULLABY, Section 4.2 presents the technique, Section 4.3 presents implementation details, Section 4.4 presents our evaluation of LULLABY Section 4.5 discusses other design choices for LULLABY, and Section 4.6 summarizes this chapter.

4.1 Examples: What Performance Bugs Have CondBreak fixes?

We discuss below two bug characteristics that help understand what performance bugs have CondBreak fixes. We call *result instruction (RI)* a loop instruction that *may* write to variables live and memory reachable after the loop. RIs are important in understanding performance bugs that have CondBreak fixes. For example, if all RIs in a loop do not (need to) execute under a certain condition, the entire loop, *including* all non-RIs, can be skipped.

(A) Where the computation is wasted: A loop-related performance bug can waste computation either in an entire iteration or in parts of an iteration, in consecutive or arbitrary iterations, in iterations at the start, end, or middle of the loop. In order to have a CondBreak fix (i.e., break out the loop under a certain condition), a performance bug has to waste computation in the entire iteration in one of the following three locations: (1) for every iteration in the loop, short as *Every*, (2) for every iteration at the end of the loop, short as *Late*, or (3) for every iteration at the start of

the loop, short as *Early*. Bugs that waste computation in category Every can be fixed by breaking out of the loop if the L-Break condition is satisfied at the loop entrance, effectively skipping the entire loop. Bugs that waste computation in category Late can be fixed by breaking out of the loop once the computation waste starts. Bugs that waste computation in category Early can be fixed by iterating from the end of the loop and breaking out of the loop once the computation waste starts. Bugs that waste computation only in parts of an iteration or only in specific iterations cannot be fixed by CondBreak fixes and are not the focus of LULLABY.

(B) How the computation is wasted: In order for the computation of an entire iteration to be wasted under a certain condition, i.e., the L-Break condition, every RI in an iteration has to fall into one of following three cases: (1) the RI is not executed under the L-Break condition, short as *No-Result*, (2) the RI is executed, but under the L-Break condition its result does not change the values used by computation after the loop, short as *Useless-Result*, or (3) the RI is executed and its result changes the values used by computation after the loop, but under the L-Break condition this result does not affect the perceived outcome of the program, short as *Semantically-Useless-Result*. Identifying Semantically-Useless-Result RIs usually requires developers' expert knowledge, and the corresponding L-Break conditions are likely difficult to express in source-code. LULLABY focuses on No-Result and Useless-Result RIs.

Based on the above discussion, a loop can have a performance bug fixed by a CondBreak fix if *all* RIs in the loop belong to one of the six types shown in Table 4.1. Note that the three computation-waste locations in (A) effectively describe which instances of an RI can be skipped in the loop. We describe individual RIs of Types 1–4, i.e., the types LULLABY focuses on, in Sections 4.1.1–4.1.4, and we present how multiple RIs appear in the same bug in Section 4.1.6. We briefly discuss Type X and Type Y in Section 4.1.5.

	Every	Late	Early
No-Result	Type 1	Type 2	Type Y
Useless-Result	Type X	Type 3	Type 4

Table 4.1: Types of RIs

All the bugs in the following examples are *previously unknown* real-world performance bugs found by LULLABY. We reported all these bugs to developers and the developers fixed all of them.

4.1.1 Type 1 RIs

Figure 4.1 shows a performance bug from Groovy containing a Type 1 RI. Line 3 is the fix and it is *not* part of the original buggy code. The only RI in this loop is `return true` (line 7), which writes the method's return value and also causes the code after the loop to not execute. When `argTypes` is initialized to a non-empty array, the RI *cannot* execute throughout the loop, the entire loop computation is wasted, and the loop can just be skipped. The reason is that `argTypes` is never modified inside the loop. When `argTypes` is initialized to a non-empty array, both `argTypes == null` and `argTypes.length == 0` (line 5) are false throughout the loop, which makes `isZeroArg` false, which in turn makes `match` (line 6) false, which in turn means the RI cannot execute.

```
1 Class[] argTypes = ...
2 for (Iterator i = methods.iterator(); i.hasNext();) {
3 + if (!(argTypes == null) && !(argTypes.length == 0)) break; // FIX
4   MethodNode mn = (MethodNode) i.next();
5   boolean isZeroArg = (argTypes == null || argTypes.length == 0);
6   boolean match =mn.getName().equals(methodName) && isZeroArg;
7   if (match) return true; // RI
8 }
```

Figure 4.1: Type 1 RI in a Groovy performance bug

This RI is of Type 1 because, if the I-Break condition is true at the start of the loop, the RI is not executed (category No-Result) in any iteration of the entire loop (category Every). The I-Break condition for the RI is that both `argTypes == null` and `argTypes.length == 0` are false. The L-Break condition is the same as the I-Break condition because there is only one RI. The CondBreak fix is the code added in line 3 (the + at the start of line 3 means the line is added), i.e., the loop breaks when the L-Break condition is true. We discuss fixes equivalent to the CondBreak fix in Section 4.2.5.

4.1.2 Type 2 RIs

Figure 4.2 shows a performance bug from PDFBox containing a Type 2 RI. There are three loops in this code and two RIs, as shown in the figure. RI 1 is an RI for all three loops because it writes `alreadyPresent`, which is live at the end of all three loops. Similarly, RI 2 is an RI for loops 1 and 2. The "..." in the figure replace some complicated control flow and method calls, which we skip for clarity. The "..." contain no RIs. In this section, we focus our discussion on RI 2 because RI 1 is of Type 3, which we will discuss in the next section. Loop 3 does not have a bug, which we will further explain in Section 4.2.4. Loops 1 and 2 are both buggy, as explained next for loop 1. Similar reasoning applies for loop 2.

```
1  boolean alreadyPresent = false;
2  while (itActualEmbeddedProperties.hasNext()) { // Loop 1
3  + if (alreadyPresent) break; // FIX
4    ... // non-RIs
5         while (itNewValues.hasNext()) { // Loop 2
6           ... // non-RIs
7           while (itOldValues.hasNext() && !alreadyPresent) { // Loop 3
8             oldVal = (TextType) itOldValues.next();
9             if(oldVal.getStringValue().equals(newVal.getStringValue())){
10              alreadyPresent = true; // RI 1
11          }}
12          if (!alreadyPresent) {
13            embeddedProp.getContainer().addProperty(newVal);// RI 2
14 }}}}}
```

Figure 4.2: Type 2 RI in a PDFBox performance bug

For the first few loop 1 iterations, `alreadyPresent` is false, the condition on line 12 evaluates to true, and RI 2 executes and performs useful computation. However, once `alreadyPresent` is set to true on line 10, the condition on line 12 remains false for the remainder of the loop, and all the remaining computation in the loop can just be skipped. The reason is that the entire loop cannot assign `alreadyPresent` to false. Consequently, once `alreadyPresent` becomes true on line 10, it remains true and disables the execution of RI 2 for the remainder of the loop.

RI 2 is of Type 2 because, once the I-Break condition becomes true, RI 2 is not executed (category No-Result) for the remaining loop iterations (category Late). The I-Break condition for RI 2 is that `alreadyPresent` equals true. We will explain in the next section that the I-Break condition for RI 1 is also that `alreadyPresent` equals true. The L-Break condition is the

conjunction of the two I-Break conditions, i.e., `alreadyPresent` equals true. The CondBreak fix is the code added in line 3, i.e., the loop breaks when the L-Break condition is true.

4.1.3 Type 3 RIs

Figure 4.3(a) shows a performance bug from Tomcat containing two Type 3 RIs, RI 1 and RI 2. Both RIs set variable `elExp` to true. Once either RI is executed, the remaining computation in the loop becomes unnecessary, at best setting `elExp` to true again.

```
1  boolean elExp = ...
2  while (nodes.hasNext()) {
3  + if (elExp) break; // FIX
4    ELNode node = nodes.next();
5    if (node instanceof ELNode.Root) {
6      if (((ELNode.Root) node).getType() == '$') {
7        elExp = true; // RI 1
8      } else if (checkDeferred && ((ELNode.Root) node).getType()=='#'
9                  && !pageInfo.isDeferredSyntaxAllowedAsLiteral() ) {
10       elExp = true; // RI 2
11 }}}
```

(a) A Type 3 RI in a Tomcat performance bug

```
1    valid &= child.validate(); // RI
```

(b) Type 3 RI in a Sling performance bug

Figure 4.3: Type 3 RIs

RI 1 is of Type 3 because, once the I-Break condition becomes true, RI 1's results for the remaining iterations (category Late) do not change the values used by future computation (category Useless-Result). Similar reasoning applies for RI 2. The I-Break condition for RI 1 is that `elExp` equals true. RI 2 has the same I-Break condition. The L-Break condition is the conjunction of the two I-Break conditions, i.e., `elExp` equals true. The CondBreak fix is the code on line 3, i.e., the loop breaks when the L-Break condition is true.

Note that, for RI 1 and RI 2 to be of Type 3, `elExp` can have any type, not necessarily boolean, as long as `elExp` is assigned a constant. In fact, even if `elExp` is not assigned a constant, there is still an alternative way to set `elExp` to a value that does not change after some time, as shown in Figure 4.3(b). In Figure 4.3(b), once `valid` is set to false, the semantics of the `&=` operator ensures `valid` remains false. The I-Break condition is that `valid` equals false.

4.1.4 Type 4 RIs

Figure 4.4(a) shows a performance bug from JMeter containing a Type 4 RI (line 6). The variable `length` keeps getting overwritten by the RI. Consequently, all iterations before the last iteration that writes `length` are wasted. The reason is that computation after the loop will only see the last value written to `length`. This last value does not depend on previous iterations, except for the value of `idx`, which can be computed when iterating from the end of the loop.

```
1 int length = ...
2 for (int idx = 0; idx < headerSize; idx++) {
3 @ for (int idx = headerSize - 1; idx ¿= 0; idx−−) { // FIX
4   Header hd = mngr.getHeader(idx);
5   if (HTTPConstants.HEADER.equalsIgnoreCase(hd.getName())) {
6     length = Integer.parseInt(hd.getValue()); // RI
7 +   break; // FIX
8 }}
```

(a) Bug and *Alternative* fix

```
1  int length = ...
2  + boolean wasExecuted = false; // FIX
3  for (int idx = 0; idx < headerSize; idx++) {
4  @ for (int idx = headerSize - 1; idx ¿= 0; idx−−) { // FIX
5  + if (wasExecuted) break; // FIX
6    Header hd = mngr.getHeader(idx);
7    if (HTTPConstants.HEADER.equalsIgnoreCase(hd.getName())) {
8  +   if (!wasExecuted) { // FIX
9  +     wasExecuted = true; // FIX
10     length = Integer.parseInt(hd.getValue()); // RI
11 +   } // FIX
12 }}
```

(b) *CondBreak* fix

Figure 4.4: Type 4 RI in a JMeter performance bug. The fix in 4.4(a) is an *Alternative* fix. The *CondBreak* fix is in 4.4(b).

The RI is of Type 4 because its results for early iterations (category Early) do not affect the values used by future computations (category Useless-Result). The I-Break condition for the RI is that `length` has been written in the loop when iterating from the end of the loop. The L-Break condition is the same as the I-Break condition because there is only one RI. For clarity, the CondBreak fix is shown separately, in Figure 4.4(b). This fix looks complex because we want to make the L-Break condition (i.e., `wasExecuted` equals true) explicit in the code. Differently from the Type 3 RI in Figure 4.3(a), the RI in this example does not set `length` to a constant. Therefore, we have to create an extra variable `wasExecuted` (lines 2, 5, 8, 9, 11) to track whether `length` has been written. Figure 4.4(a) shows a simpler, *alternative* fix, that does not use `wasExecuted`. In the alternative fix, the reversed loop breaks the first time when the RI is executed (line7). The simpler, alternative fix comes at a price: it is correct only when the loop has one RI. Otherwise, breaking out of the loop after one RI would incorrectly miss the execution of remaining RIs.

4.1.5 Type X and Type Y RIs

LULLABY checks for Type X RIs when it checks for Type 3 RIs. For example, in Figure 4.3(a), if the value of `elExp` was a constant true before the loop started, then RI 1 and RI 2 would be Type X. In practice, we never encountered bugs containing Type X RIs and it also seems unlikely developers would purposefully write an instruction that cannot change a variable's value. A Type Y RI cannot write to the same memory locations in different loop iterations. Otherwise, the CondBreak fix, which requires reversing the loop, would incorrectly change the program semantic. Checking that different instances of an instruction can only write to different memory locations requires complex static analysis, and LULLABY does not perform such checks.

4.1.6 Bugs with Multiple RIs

A buggy loop can contain multiple RIs of the same or different types. The only constraint is that a Type 4 RI cannot co-exist with Type 2 or Type 3 RIs because the former requires the bug fix to skip iterations at the start of the loop and the latter requires the bug fix to skip iterations at the end of the loop. In practice, we did not encounter Type 1 RIs co-existing with other types of RIs. Some RIs can be of multiple types, as shown next for RI 3 and RI 9. For loops that contain multiple RIs, the L-Break condition is the conjunction of these RIs' I-Break conditions.

Figure 4.5 shows an example bug with multiple RIs from PDFBox. Unlike the bugs in Figure 4.2 and Figure 4.3(a), this bug has nine RIs that have different I-Break conditions. RI 1 and RI 2 are Type 2 RIs because once `annotNotFound` is set to false (line 9), `annotNotFound` *cannot become true again*, the condition on line 6 always evaluates to false, and therefore RI 1 and RI 2 cannot be executed in the remaining loop iterations. Their I-Break conditions are both `annotNotFound` equals false. Similar reasoning applies for RI 4–8, `sigFieldNotFound` (line 14), and the condition on line 11. RI 3 and RI 9 are *simultaneously* Type 2 (similarly to RI 1–2 and RI 4–8, respectively) and Type 3. Their I-Break conditions are `annotNotFound` equals false and `sigFieldNotFound` equals false. The L-Break condition is the conjunction of all nine I-Break

conditions, i.e., both `annotNotFound` and `sigFieldNotFound` equal false. Once *both* `annotNotFound` and `sigFieldNotFound` become false, the loop cannot produce any result and all the remaining iterations are unnecessary. The CondBreak fix for this bug (line 4) is similar to the fixes for the previous bugs, i.e., the loop breaks when the L-Break condition is true.

```
1 boolean annotNotFound = ...
2 boolean sigFieldNotFound = ...
3 for ( COSObject cosObject : cosObjects ) {
4 + if (!annotNotFound && !sigFieldNotFound) break; // FIX
5   ... // some non−RIs
6   if (annotNotFound && COSName.ANNOT.equals(type)) {
7     ... // RI 1 and some non-RIs
8     signatureField.getWidget().setRectangle(rect); // RI 2
9     annotNotFound = false; // RI 3
10  }
11  if (sigFieldNotFound && COSName.SIG.equals(ft)&&apDict!=null){
12    ... // RI 4, RI 5, RI 6, RI 7 and some non-RIs
13    acroFormDict.setItem(COSName.DR, dr); // RI 8
14    sigFieldNotFound=false; // RI 9
15 }}
```

Figure 4.5: Multiple RIs in a PDFBox performance bug

4.2 Detecting and Fixing Bugs that Have CondBreak Fixes

We first present the LULLABY high-level algorithm (Section 4.2.1) and then we discuss the algorithm steps in detail (Sections 4.2.2–4.2.5).

4.2.1 High-Level Algorithm

Figure 4.6 shows the high-level algorithm for LULLABY. LULLABY is a static technique that works on intermediate code representation (IR). LULLABY receives as input the loop to analyze and various information to help the static analysis, e.g., the control flow graph for the method containing the loop, pointer aliasing information, and a call graph.

LULLABY works in five steps. First, LULLABY computes the loop RIs using routine static analysis (line 2). Second, for each RI `r`, LULLABY checks if `r` belongs to one of the four types presented in Section 4.1 and computes `r`'s I-Break condition accordingly (lines 4–10). If `r` does not belong to any of the four types, all the conditions computed on lines 5–8 are false and therefore

```
1 void detectPerformanceBug(Loop l, Method m, AliasInfo alias) {
2   Set⟨Instruction⟩ allRIs = getRIs(l, m, alias);
3   Set⟨Condition⟩ allCond = new Set⟨Condition⟩();
4   for (Instruction r : allRIs) {
5     Condition one = typeOne(r, l, m, alias);
6     Condition two = typeTwo(r, l, m, alias);
7     Condition three = typeThree(r, l, m, alias);
8     Condition four = typeFour(r, l, m, alias, allRIs.size());
9     if(one.false()&&two.false()&&three.false()&&four.false()) return;
10    allCond.putIfNotFalse(one, two, three, four);
11  }
12  if(satisfiedTogether(allCond) && notAlreadyAvoided(allCond, l)) {
13    String fix = generateFix(allCond, l, m);
14    reportBugAndFix(fix, allRIs, allCond);
15 }}
```

Figure 4.6: LULLABY high-level algorithm

the loop does not have a bug (line 9). If `r` is of one of the four types, LULLABY saves for further use `r`'s I-Break condition (line 10). Third, LULLABY checks if all RIs can be skipped *simultaneously* without changing the program outcome, i.e., if the I-Break conditions for individual RIs can be satisfied simultaneously (line 12). The conjunction of the I-Break conditions is the L-Break condition. Fourth, LULLABY checks if the computation waste in the loop is not already avoided, i.e., if the loop does not already terminate when the L-Break condition is satisfied (line 12). Fifth, using the L-Break condition, LULLABY generates a fix (line 13) and reports the bug (line 14). The bug report contains the fix, and, for each RI, the RI type and I-Break condition.

The above algorithm enables LULLABY to detect and fix performance bugs that involve multiple RIs, either of the same or different types, similar to the examples in Figures 4.2, 4.3(a), and 4.5. This is because, after step two, LULLABY works only with a collection of conditions, and LULLABY is not concerned with how these conditions were obtained in step two.

Preliminary: Boolean Expressions

To compute the I-Break conditions and the L-Break condition, LULLABY reasons about boolean expressions. LULLABY represents and reasons about a boolean expression as one or multiple *Atoms* connected by boolean operators (`NOT`, `AND`, `OR`). An Atom refers to either a boolean variable or a boolean expression containing non-boolean operators. Atoms do not contain other Atoms. For example, an Atom could be a method call returning a boolean value or a comparison between two

integers. To keep complexity low and scale, LULLABY does not reason about operations inside Atoms. An Atom can be either true or false but *not* both simultaneously.

For space limitations, we do not go into the details of how LULLABY works with boolean expressions, and we give only a high-level overview for two techniques used by LULLABY. These two techniques can be substituted by more sophisticated techniques, such as symbolic execution. However, for LULLABY's purposes, these two techniques offer good results at considerably reduced complexity. Technique **T-PathExec** computes the execution condition of a loop instruction as the disjunction of all path constraints that correspond to the acyclic execution paths leading from the loop header to the instruction. A path constraint is the conjunction of all branch conditions, represented by Atoms and negated when necessary, along a path. LULLABY uses T-PathExec in steps two and four of the LULLABY algorithm. Technique **T-Instantiation** computes, for a boolean expression `E` (in DNF form) and some Atoms `set`, the values of the `set` Atoms for which `E` is guaranteed false or true. Conversely, T-Instantiation determines if `E` *may* be true or false, irrespective of Atoms in `set`. T-Instantiation tries all possible combinations of values for the `set` Atoms and uses logic rules such as "False `AND` Unknown equals False" to determine the value of `E`. For example, for `E = $Atom1 AND $Atom2` and `set = {$Atom1}`, T-Instantiation determines that, when `$Atom1` equals true, `E` is Unknown, and, when `$Atom1` equals false, `E` is false. In the usage context of LULLABY, `set` has few Atoms (e.g., when identifying Type 1 RIs, `set` contains the loop-invariant Atoms in `E`, which is rarely more than 3), and therefore T-Instantiation rarely tries more than 8 combinations. LULLABY uses T-Instantiation in steps two, three, and four of the LULLABY algorithm.

4.2.2 Detecting the Four RI Types

In the second step of the LULLABY algorithm, LULLABY determines if a given RI (all RIs are known from step one) belongs to one of the four types and, if so, the RI's I-Break condition. Identifying all RIs that belong to each type would require complicated and non-scalable analysis. At the same time, not all RIs that belong to the four types are common and have I-Break conditions that

are easy to express in source-code. Our design intentionally makes LULLABY focus on RIs whose type can be identified using scalable analysis and whose I-Break conditions are easy to express in source-code. LULLABY can miss some RIs that belong to these four types, as explained below. We next describe each type and the algorithm LULLABY uses to detect the type (Sections 4.2.2–4.2.2).

Type 1 RIs

An RI `r` is of Type 1 if there exists a condition `C` such that `r` cannot execute throughout the loop if `C` is true when the loop starts. The I-Break condition for `r` is `C`. To judge whether `r` belongs to Type 1, LULLABY searches for `r`'s I-Break condition. **Theoretically**, the I-Break condition could be composed of any variables and expressions that appear or do not appear in the entire program. However, inferring or searching for such generic I-Break conditions is difficult. **In practice**, LULLABY considers only I-Break conditions that (1) can be computed by analyzing the potential execution paths that may reach `r` from the loop header and (2) are composed of Atoms that can be proved to be loop-invariant based on control and data-flow analysis. Constraint (1) makes detecting candidate I-Break conditions feasible and scalable, while constraint (2) makes it easy to prove that a candidate I-Break condition cannot change its value throughout the loop execution. Additionally, constraint (1) ensures the I-Break condition is easy to express in source-code, because the candidate I-Break conditions contain only variables and expressions already present in the loop.

Figure 4.7 shows the algorithm for detecting if an RI `r` is of Type 1 and, if so, `r`'s I-Break condition. LULLABY uses T-PathExec to compute the execution condition `rCond` for `r` (lines 3, 4), gets the loop-invariant Atoms in `rCond` (line 5), uses T-Instantiation to get these Atoms' values for which `rCond` is guaranteed to be false irrespective of the values of other Atoms in `rCond` (lines 6–9), and constructs `r`'s I-Break condition based on these Atom values (lines 9, 10). For example, for the RI in Figure 4.1, LULLABY identifies three Atoms that are part of the RI's execution condition in the loop: `$Atom1 = (argTypes == null)`, `$Atom2 = (argTypes.length == 0)`, and `$Atom3 = (mn.getName().equals(methodName))`. LULLABY determines that `$Atom1` and

$Atom2 are loop-invariant, determines that RI's execution condition is guaranteed to be false when $Atom1 and $Atom2 are both false, irrespective of the value of $Atom3, and computes the RI's I-Break condition as both $Atom1 and $Atom2 being false.

```
1 ORCond typeOne(Instruction r, Loop l, Method m, AliasInfo alias) {
2   ORCond res = new ORCond(false);
3   Set⟨List⟨Pair⟨CondIns,Boolean⟩⟩⟩paths = acyclicPaths(l.head(), r, l);
4   ORCond⟨ANDCond⟩ rCond = fromPathsToCond(paths);
5   Set⟨Atom⟩ invarAtomsInCond = getLoopInvariantAtoms(rCond);
6   List⟨List⟨InstanceCond⟩⟩perms = permutations(invarAtomsInCond);
7   for (List⟨InstanceCond⟩ curPerm : perms)
8     if (condIsNegated(rCond, curPerm))
9       res.add(new ANDCond(curPerm));
10  return res;
11 }
```

Figure 4.7: Detecting if an RI is of Type 1

Type 2 RIs

An RI r is of Type 2 if there exists a condition C such that r cannot execute once C becomes true during the loop execution. The I-Break condition for r is C. Detecting Type 2 and Type 1 RIs are similar and have similar challenges. LULLABY applies similar constraints when searching for r's I-Break condition, with only one difference for constraint (2). For Type 2 checking, LULLABY only considers Atoms that are assigned one constant in the loop. This constraint makes it easy to prove that a candidate I-Break condition cannot change its value after all its component Atoms are updated during the loop execution.

The algorithm for detecting if an RI is of Type 2 is similar to the algorithm in Figure 4.7, with two modifications. First, instead of identifying loop-invariant Atoms in rCond (line 5, Figure 4.7), the Type 2 algorithm detects Atoms in rCond that are assigned only one boolean constant value in the loop. For example, for the bug in Figure 4.2, LULLABY detects that alreadyPresent is assigned true on line 10. Second, when LULLABY computes the Atoms' values for which rCond is guaranteed to be false (lines 6–9, Figure 4.7), LULLABY takes into account that the Atoms identified above can take only the corresponding constant values. For example, alreadyPresent can become true but not false.

Type 3 RIs

An RI `r` is of Type 3 if, after a certain loop iteration, `r` can only write to the same output locations it wrote in previous iterations and the values written are identical to the existing values in these locations; we call these existing values `S`. The I-Break condition is that the output locations contain `S`. To judge whether `r` belongs to Type 3, LULLABY examines `r`'s output locations and output values. **Theoretically**, `r` could be any instruction, including a call to a method with complex control flow that writes to many memory locations. Reasoning about such a general `r` is difficult. **In practice**, LULLABY focuses only on RIs that (1) have a single output location, (2) either write a constant (similar to Figure 4.3(a)) or perform the `&=` or `|=` operations (similar to Figure 4.3(b)), which effectively correspond to `S` being constants false or true, respectively, and (3) have an output location that is not written to in the loop with other values except `S`. Constraint (1) makes it easy to detect `r` does not change its output locations, while constraints (2) and (3) make it easy to prove that, after a certain loop iteration, `r` can only write `S`. Additionally, constraint (2) ensures the I-Break condition is easy to express in source-code, because `S` can be identified statically.

The algorithm for detecting if an RI is of Type 3 follows the above description. LULLABY computes the left-hand side for the RI. If the location of the left-hand side can change, the RI is not of Type 3. LULLABY also checks if the left-hand side is assigned only one constant or contains either the `&=` or the `|=` operator. The I-Break condition is that the left-hand side equals the constant value or equals false or true, respectively. For example, for the bug in Figure 4.3(a), for `RI 1`, LULLABY detects that `elExp` can be assigned only true in the loop and therefore the I-Break condition is that `elExp` equals true. For the bug in Figure 4.3(b), LULLABY detects the RI contains the `&=` operator and the I-Break condition is that `valid` equals false.

Type 4 RIs

An RI `r` is of Type 4 if `r` outputs values independent of computation in early iterations, except for the loop index computation, and if `r` cannot change its output locations. The I-Break condition is

that the values in the output locations have been updated the first time when iterating from the end of the loop. To judge whether `r` belongs to Type 4, LULLABY examines `r`'s output locations and output values. **Theoretically**, `r` could be any instruction, including a method call, and checking the above conditions for such a general `r` is difficult. **In practice**, LULLABY focuses only on RIs that (1) have a single output location, (2) appear in loops that have no cross-iteration data dependency, except for the loop index computation, and (3) appear in loops that have only one RI. Constraint (1) makes it easy to detect that `r` does not change its output locations and constraint (2) makes it easy to prove that the output values are independent of computation in earlier iterations. Additionally, constraint (3) ensures the fix is similar to the alternative fix in Figure 4.4(a), instead of the CondBreak fix in Figure 4.4(b).

The algorithm for detecting if an RI is of Type 4 checks if the loop has one RI and if the loop has no data dependencies across iterations except for the loop index. If the checks succeed, the RI is of Type 4.

4.2.3 Checking RIs can be Skipped Simultaneously

In the third step of the LULLABY algorithm, LULLABY checks if a scenario exists for which all RIs can be *simultaneously* skipped without changing the program outcome, i.e., all RIs' I-Break conditions can be satisfied simultaneously. The L-Break condition enabling this scenario is the conjunction of all I-Break conditions.

Figure 4.8 gives a simplified example from Lucene of why LULLABY performs this check. The only two RIs in this loop are both of Type 1 and have I-Break conditions `roundNum < 0` equals true and `roundNum < 0` equals false, respectively. However, this loop does not contain a performance bug because the executions of the two RIs cannot be skipped *simultaneously*, i.e., `roundNum < 0` cannot simultaneously be true and false.

To perform this check, LULLABY applies T-Instantiation on the conjunction of selected I-Break conditions, effectively checking there is at least one combination of the involved Atoms' values that makes the conjunction true. LULLABY optimizes this check by applying it on selected, instead

```
1 int roundNum = ...
2 StringBuilder sb = ...;
3 for (final String name : colForValByRound.keySet()) {
4   if (roundNum < 0) {
5     sb.append(Format.formatPaddLeft("-", template)); // RI 1
6   } else {
7     sb.append(Format.format(ai[n], template)); // RI 2
8 }}
```

Figure 4.8: Simplified code from Lucene. The RIs are of Type 1, but these RIs do not create a performance bug.

of all I-Break conditions, because the definitions of some RI types already guarantee that their corresponding I-Break conditions will never conflict with each other. For example, the I-Break condition for a Type 3 RI cannot conflict with the I-Break condition for a Type 1 RI. The reason is that the Atoms in the I-Break condition of a Type 1 RI must be loop-invariant and therefore cannot appear in a Type 3 RI's I-Break condition.

4.2.4 Checking the Computation Waste is Not Already Avoided

In the fourth step of the LULLABY algorithm, LULLABY checks the execution is not already exiting the loop when the L-Break condition is true. Loop 3 in Figure 4.2 (lines 7–11) is an example of why LULLABY performs this check. As mentioned in Section 4.1.2, loop 3 does not have a performance bug and we now explain why. The only RI in loop 3 is RI 1 (line 10), which is of Type 3 and therefore it may seem loop 3 performs useless computation once `alreadyPresent` is set to true. However, once `alreadyPresent` becomes true, the loop exits (`!alreadyPresent`, line 7), and therefore the loop does not have a performance bug. LULLABY performs this check using T-PathExec, which is used to detect the execution condition for loop exits, and T-Instantiation, which is used to detect if the paths to loop exits may be taken in the original code when the L-Break condition is true.

4.2.5 Automatic Fix Generation

In the fifth step of the LULLABY algorithm, LULLABY generates source-code fixes. Automatic bug fixing is a difficult problem in general, but it is feasible for LULLABY because LULLABY

focuses on bugs that have CondBreak fixes.

LULLABY generates fixes in two steps. First, LULLABY generates a source-code level L-Break condition composed of source-code level variables declared and initialized outside of the loop. Because the variable and method names are available at the IR level, this process is straightforward in general. The only challenge is that some Atoms in the L-Break condition may involve variables that are not suitable for the final source-code fix. Specifically, some variables are introduced by compiler in the IR representation and do not exist in the source-code, while some other variables are declared or initialized inside the loop by developers and therefore cannot be used in the fix at the start of the loop. The solution is straightforward: LULLABY repetitively replaces these unsuitable variables with their assigned expression. This step guarantees not to change the value of the L-Break condition because of the way the I-Break conditions are defined in Section 4.2.2.

Second, LULLABY formats the fix according to the types of the RIs, and computes the line number where the fix is to be inserted using line number information from the IR. When the loop contains RIs of Type 1, 2, or 3, the fix simply inserts `if (L-Break condition) break` after the loop header, as shown in the Section 4.1 examples. In the special case when the loop contains *only* Type 1 RIs, the fix is `if (L-Break condition == false) theLoop`, effectively executing the loop only when the L-Break condition is false. This alternative fix is equivalent with the CondBreak fix, but is preferred by developers. When the loop contains a Type 4 RI, LULLABY reverses the loop if the loop has an integer index variable that is incremented by one in the loop header, similar to that in Figure 4.4(a); otherwise, LULLABY reports fix generation failure. General loop reversal is difficult to do automatically, but treating the above case was enough to fix the bugs we encountered in practice. LULLABY can handle more cases for loop reversal in the future. We discuss this and a second reason why LULLABY may fail to generate fixes (imprecision in the static analysis framework) in Section 4.4.3. In practice, LULLABY generated correct fixes for 149 out of 150 bugs.

4.3 Two Implementations

We implement *two* LULLABY tools, for both Java and C/C++ programs, which we call LULLABY-J and LULLABY-C, respectively. We implement LULLABY-J and LULLABY-C using WALA [2] and LLVM [44] static analysis frameworks, respectively. Implementing the high-level algorithms in Section 4.2.2 takes into account the fact that the IRs provided by WALA and LLVM are in SSA form. LULLABY-J uses the pointer aliasing information provided by WALA. LULLABY-C conservatively assumes that every write to heap is an RI, and therefore LULLABY-C can label many non-RI instructions as RI. This can cause LULLABY-C to miss some bugs, because the spurious RIs can make unnecessary loop computation look useful. However, in practice, this was not a major problem, as LULLABY-C found 89 new bugs in widely used C/C++ applications. The analysis in both LULLABY-J and LULLABY-C is inter-procedural. The implementation closely follows the presentation in the previous section. The only exception is that LULLABY-C currently detects only bugs that have one RI, and therefore LULLABY-C does not perform step three in the LULLABY algorithm. We do not discuss further implementation details due to space limitations.

4.4 Evaluation

We evaluate LULLABY on *real-world applications* from Java and C/C++ using our two LULLABY implementations, LULLABY-J and LULLABY-C, respectively. We use 11 popular Java applications (Ant, Groovy, JMeter, Log4J, Lucene, PDFBox, Sling, Solr, Struts, Tika, and Tomcat) and 4 widely used C/C++ desktop and server applications (Chromium, GCC, Mozilla, and MySQL). Table 4.2 gives a short description of these applications. We analyze the latest code versions of these applications, except for Lucene, for which we use a slightly older version, because LULLABY does not support Java 7. Of all the Lucene bugs found by LULLABY, only two bugs are in code that no longer exists in the latest version. In total, LULLABY generates 173 bug reports. This section first presents the 150 new bugs found by LULLABY. It then discusses the 23 false positives reported by LULLABY, the fix generation results, and LULLABY's running time. We conduct the experiments

on two Intel i7, 4-core, 8 GB machines, running at 2.5 GHz and 3.4 GHz for the Java and C/C++ experiments, respectively.

L	#	App	Description	LoC	Classes(J) Files(C)
Java	1	Ant	build tool	140,674	1,298
	2	Groovy	dynamic language	161,487	9,582
	3	JMeter	load testing tool	114,645	1,189
	4	Log4J	logging framawork	51,936	1,420
	5	Lucene	text search engine	441,649	5,814
	6	PDFBox	PDF framework	108,796	1,081
	7	Sling	web app. framework	202,171	2,268
	8	Solr	search server	176,937	2,304
	9	Struts	web app. framework	175,026	2,752
	10	Tika	content extraction	50,503	717
	11	Tomcat	web server	295,223	2,473
C/C++	12	Chromium	web browser	13,371,208	10,951
	13	GCC	compiler	1,445,425	781
	14	Mozilla	web browser	5,893,397	5,725
	15	MySQL	database server	1,774,926	1,684

Table 4.2: The applications used in experiments

4.4.1 New Bugs Found by LULLABY

LULLABY is very effective at detecting performance bugs. LULLABY finds a *total of 150 new bugs*, 61 bugs in Java applications and 89 bugs C/C++ applications. Of these, *116 bugs*, 51 and 65 in Java and C/C++, respectively, *have already been fixed by developers*. Of the bugs not yet fixed, 7 bugs are confirmed and still under consideration by developers and 16 bugs are still open. 7 bugs were not fixed because they are in deprecated code, old code, test code, or auxiliary projects. *Only 3 bugs* were not fixed because developers considered that the bugs do not have a significant performance impact. *Only 1 bug* was not fixed because developers considered that the fix hurts code readability.

We manually inspected all Java and C/C++ bugs reported by LULLABY and we find they are similar. The only exception is that LULLABY-C can currently detect only bugs with one RI (Section 4.3), and therefore all the C/C++ bugs in this evaluation have one RI. The bug examples shown

so far in the chapter are from Java code. Figure 4.9 shows an example bug from GCC. This bug was confirmed and fixed by developers. In this bug, `FOR_EACH_EDGE` (line 2) is a C/C++ macro that expands to a loop over `e->src->succs`. This loop has a Type 3 RI (line 7), and all loop iterations after the RI sets `irred_invalidated` to true waste computation. The I-Break condition and the L-Break condition are that `irred_invalidated` equals true, and the CondBreak fix is the code added in line 3.

```
1 bool irred_invalidated = ...
2 FOR_EACH_EDGE (ae, ei, e->src->succs) {
3 + if (irred_invalidated) break; // FIX
4   if (ae != e && ae->dest != EXIT_BLOCK_PTR
5           && !bitmap_bit_p (seen, ae->dest->index)
6           && ae->flags & EDGE_IRREDUCIBLE_LOOP) {
7     irred_invalidated = true; // RI
8 }}
```

Figure 4.9: A GCC performance bug found by LULLABY

Table 4.3 shows the detailed results for the new bugs found by LULLABY. The numbers in the table refer to the numbers of distinct buggy loops, with each loop containing one or multiple RIs. 16 out of the 61 Java bugs in Table 4.3 contain more than one RI. As explained in Section 4.1.6, most of these 16 bugs contain RIs of the same type, with only a few bugs containing RIs of Type 2 and Type 3, as shown in the table (the column headers show the type of the RIs in the bug). LULLABY-C can currently detect only bugs with one RI (Section 4.3), and therefore no C/C++ bug in Table 4.3 contains multiple RIs.

LULLABY found bugs in all 15 applications in Table 4.3, including in GCC, which is highly tuned for performance and has been developed for more than two and a half decades. Indeed, *all the bugs that we reported to GCC have already been fixed by developers*.

LULLABY found all four RI types in bugs. Looking at the type breakdown in Table 4.3, we see Type 3 RIs are more frequent than RIs of other types. We manually inspect all the bugs reported by LULLABY and we find the bugs containing Type 3 RIs typically appear in code performing a linear search for objects that have certain properties, such as the bugs in Figure 4.3. This is a common operation in real-world code, and therefore it presents more opportunities for developers to introduce such bugs.

Application	Type 1 RIs	Type 2+3 RIs	Type 3 RIs	Type 4 RIs	**SUM**
Ant	0	0	1	0	**1**
Groovy	2	0	7	0	**9**
JMeter	0	0	3	1	**4**
Log4J	0	0	5	1	**6**
Lucene	6	0	7	1	**14**
PDFBox	1	5	3	1	**10**
Sling	0	0	6	0	**6**
Solr	0	0	2	0	**2**
Struts	2	0	2	0	**4**
Tika	0	0	1	0	**1**
Tomcat	0	0	3	1	**4**
Chromium	0	0	13	9	**22**
GCC	1	0	21	0	**22**
Mozilla	0	0	20	7	**27**
MySQL	3	0	13	2	**18**
SUM:	**15**	**5**	**107**	**23**	**150**

Table 4.3: New bugs found by LULLABY

4.4.2 False Positives

LULLABY reports few false positives, as shown in Table 4.4. We manually inspect all false positives and find three causes. *Complex Analysis* false positives occur when LULLABY incorrectly judges, in step three of its algorithm, that some L-Break conditions are satisfiable. Such false positives can be reduced by complex analysis, as described in this section, but the number of such false positives does not justify the added complexity. *Concurrent* false positives are caused by expressions that appear to be loop-invariant, but that in reality can be modified by code in a concurrent thread. Such false positives can be reduced using static analysis [58] or heuristics [83]. *Infrastructure* false positives are caused by two limitations in the WALA static analysis framework, described in this section.

Figure 4.10 shows a Complex Analysis false positive from Tomcat. Here, LULLABY detects that RI 1 and RI 2 are of Type 1 with I-Break conditions `allRolesMode == AllRolesMode.AUTH_ONLY_MODE` equals false (line 5) and `allRolesMode == AllRolesMode.STRICT_AUTH_ONLY_MODE` equals false (line 9), respectively. LULLABY incorrectly judges, in step three of its al-

Application	Complex Aly.	Concurrent	Infrastructure
Ant	0	1	0
Groovy	0	0	0
JMeter	0	0	0
Log4J	0	2	0
Lucene	2	3	0
PDFBox	0	0	1
Sling	0	0	1
Solr	0	0	1
Struts	1	0	1
Tika	2	0	0
Tomcat	1	0	3
Chromium	0	0	0
GCC	1	0	0
Mozilla	2	0	0
MySQL	1	0	0
SUM:	**10**	**6**	**7**

Table 4.4: False positives and their cause

gorithm, that it is possible to satisfy these two I-Break conditions simultaneously, and LULLABY reports a bug. However, this conclusion is wrong because `AllRolesMode` is an enumeration with three values and the loop is executed only when `allRolesMode` is not equal to the third value. Therefore, when the loop executes, `allRolesMode` can only have one of the two remaining values. LULLABY could avoid this false positive by employing a complex analysis that takes into account the values enumeration variables can take, *and* that determines the condition under which the entire loop is executed.

```
1  AllRolesMode allRolesMode = ...;
2  for (int i = 0; i < constraints.length; i++) {
3    SecurityConstraint constraint = constraints[i];
4    if (constraint.getAllRoles()) {
5      if (allRolesMode == AllRolesMode.AUTH_ONLY_MODE) {
6        log.debug("Granting access for ..."); // RI 1
7      }
8      String[] roles = request.getContext().findSecurityRoles();
9      if (roles.length == 0 && allRolesMode == AllRolesMode.STRICT_AUTH_ONLY_MODE) {
10       log.debug("Granting access for ..."); // RI 2
11 }}}
```

Figure 4.10: Complex Analysis false positive from Tomcat

Figure 4.11 shows a Concurrent false positive from Lucene. Here, LULLABY detects Atoms `channel.isClosed()` and `thread == null` (lines 1, 2) as loop-invariant, and therefore con-

cludes that the RI on line 2 is of Type 1. However, this conclusion is wrong because `channel` and `thread` are both shared variables that can be modified by another thread, in parallel with this loop's execution. Consequently, these two Atoms are not really loop-invariant. This loop is used to block the local thread until another thread closes the channel or sets the `thread` variable to `null`. This is a typical custom synchronization that can be detected by existing tools [83].

```
1 while (!channel.isClosed()) {
2   if (thread == null) return; // RI
3   try {sleep(RETRY_INTERVAL);} catch (Exception e) {/*ignored*/}
4 }
```

Figure 4.11: Concurrent false positive from Lucene

The Infrastructure false positives are caused by two WALA limitations. First, to scale to large programs, we instruct WALA to ignore some libraries, e.g., `java.awt` and `javax.swing`, as recommended in WALA's performance guidelines [2]. This may cause WALA to give unsound results, which may cause LULLABY to miss some RIs in step one of the LULLABY algorithm. Second, WALA gives unsound results when the entry method for the analysis (i.e., the method containing the loop) has parameters of types that do not have default constructors. Similar to the first limitation, these unsound results create false positives. Such false positives may be avoided by annotating methods in the ignored libraries with effect summaries or by improving WALA's handling of entry methods.

4.4.3 Automatic Fix Generation

LULLABY successfully generates fixes for 149 out of 150 bugs. We manually inspected all these fixes and confirmed they are correct. For one bug in Tomcat, LULLABY-J could not generate a fix due to a limitation in WALA. Specifically, WALA does not always provide line number information for assignment instructions. Therefore, for this Tomcat bug, LULLABY could not generate a fix like the fix in Figure 4.4(a), because LULLABY did not know where to insert the `break`. For the other bugs involving Type 4 RIs, WALA did not suffer from this problem and LULLABY could generate fixes. Note that loop headers are *not* assignment instructions. Therefore,

generating fixes immediately before or after loop headers, which is how Lullaby generates fixes for loops containing other RI types (Section 4.2.5), is not affected by this limitation.

We compare the fixes generated by LULLABY with the fixes adopted by developers and find they are similar, with one exception. For bugs containing only one Type 3 RI, CondBreak fixes are different from manual fixes, because developers prefer to insert a `break` immediately after the RI. LULLABY could have easily followed developers' style and generated the same fixes, if WALA was able to provide the line number of the RI. However, as describe above, WALA cannot guarantee to provide line number for assignment instructions, and LULLABY chooses to generate the basic CondBreak fixes, inserted right after the loop header.

LULLABY can generate an incorrect fix for a real bug for two reasons, though in practice LULLABY did not generate any incorrect fix. First, to enable WALA to scale, LULLABY may not detect some RIs (Section 4.4.2). If this happens, the L-Break condition does not contain all the I-Break conditions and the fix causes the execution to exit the loop too early. Second, for bugs containing a Type 4 RI, not all loops can easily be reversed, as discussed in Section 4.2.5.

4.4.4 Overhead

Table 4.5 shows LULLABY's running time in minutes. Columns *Sequential* and *Parallel* give the time for the sequential and parallel version LULLABY using three threads[1], respectively. LULLABY's parallel version divides the loops in N groups, starts N threads, and lets each thread analyze the loops in one group. LULLABY-J's parallel execution takes up to two hours, for all but three applications. Most of this time is spent in WALA's inter-procedural pointer analysis. We consider this running time acceptable, because developers do not need to write test code, like for a dynamic bug detection technique, or devise complex usage scenarios, like for a profiler. Furthermore, after the initial run, subsequent runs of LULLABY on the same code can focus only on code that has changed, in the spirit of regression testing [88]. The speedup of the parallel

[1]We also ran LULLABY with four threads, but the running time is slightly higher due to machine resource contention.

version over the sequential version is over 2.5X for all but four applications, which shows LULLABY makes effective use of modern multi-core machines. LULLABY-C is much faster than LULLABY-J because LULLABY-C conservatively assumes that every write to heap is an RI. In contrast, to detect RIs, LULLABY-J uses WALA's pointer-alias analysis, which is slow. We did not consider necessary to parallelize LULLABY-C because the running time is small, ranging from several minutes for GCC and MySQL up to one and a half hours for Chromium.

Application	Sequential	Parallel	Speedup (X)
Ant	183	72	2.54
Groovy	345	128	2.70
JMeter	118	52	2.27
Log4J	108	45	2.40
Lucene	1068	417	2.56
PDFBox	106	38	2.79
Sling	355	190	1.87
Solr	1062	627	1.69
Struts	226	77	2.94
Tika	113	42	2.69
Tomcat	258	89	2.90
Chromium	85	n/a	n/a
GCC	3	n/a	n/a
Mozilla	52	n/a	n/a
MySQL	10	n/a	n/a

Table 4.5: LULLABY running time (minutes)

4.5 Discussion

Other Performance Bugs that have Non-Intrusive Fixes: LULLABY makes a first step in detecting performance bugs that have simple and non-intrusive fixes. Of course, there is still much work that needs to be done. For example, performance bugs outside loops or bugs that do not have CondBreak fixes could still have non-intrusive fixes, and they are out of LULLABY's scope. We hope LULLABY's promising results will motivate future research to detect more performance bugs that have simple and non-intrusive fixes.

Estimating the Offered Speedup: LULLABY is a static technique and cannot easily estimate

the speedup offered by the bug fix. Developers may appreciate such additional information. However, as our results show, developers typically fix the bugs reported by LULLABY, even if the exact speedup is not available. Future work can try to estimate the number of loop iterations or executed instructions based on the input size or input values [15, 30].

Other RIs that Belong to the Four Types: The gap between what RIs belong to each type in theory and what RIs LULLABY detects in practice (Section 4.2.2) may be reduced by future work. However, as discussed in Section 4.2.2, such future work needs to carefully consider the scalability of the employed analysis. Because the RIs LULLABY currently detects provided good results, we did not focus on this line of work.

4.6 Summary

Performance bugs affect even well tested software written by expert programmers. In practice, fixing a performance bug can have both benefits and drawbacks, and developers fix a performance bug only when the benefits outweigh the drawbacks. Unfortunately, the benefits and drawbacks can be difficult to assess accurately. This chapter presented LULLABY, a novel technique that detects and fixes performance bugs that have non-intrusive fixes likely to be adopted by developers. Specifically, LULLABY detects performance bugs that have CondBreak fixes: when a condition becomes true during loop execution, just break out of the loop. We evaluated LULLABY on *real-world applications*, including 11 popular Java applications (Ant, Groovy, JMeter, Log4J, Lucene, PDFBox, Sling, Solr, Struts, Tika, and Tomcat) and 4 widely used C/C++ applications (Chromium, GCC, Mozilla, and MySQL). LULLABY found *61 new performance bugs* in the Java applications and *89 new performance bugs* in the C/C++ applications. Of these bugs, *developers have already fixed 51 performance bugs* in the Java applications and *65 performance bugs* in the C/C++ applications. LULLABY makes a solid first step in detecting performance bugs that have non-intrusive fixes.

Chapter 5

Related Work

We next describe the work related to our study of performance bugs, TODDLER, and LULLABY.

5.1 Study of Performance Bugs

To the best of our knowledge, our RQ1 has not been studied before; and we discuss how our RQ2 and RQ3 are different from the related work below.

Empirical Studies of Performance Bugs: Zaman et al. [91] study security, performance, and generic bugs in the Firefox web browser. Their analysis includes metrics similar to the metrics that we use to answer RQ2. In addition, our analysis discriminates between uni-patch and multi-patch bugs, considers initial and supplementary patches, studies more applications, and analyzes additional data such as clones. Their followup paper [92] studies the bug reports for performance and non-performance bugs in Firefox and Chrome. They study how users perceive the bugs, how the bugs were reported, what developers discussed about the bug causes and the bug patches. Similar to our data in Table 2.10, they also analyze bug reports that have an input attached. Unlike their study, our study analyzes different information from bug reports, analyzes patches, differentiates between uni-patch and multi-patch bugs, and studies more applications.

Jin et al. [34] study the root cause of 109 performance bugs from five code bases, observe frequent code patterns related to performance bugs, and use these patterns to detect new performance bugs. Unlike their study, our study focuses on how performance bugs are discovered, reported, and fixed.

Empirical Studies of Generic Bugs: There are many projects that study and characterize

different aspects of generic bugs, e.g., [6, 7, 16, 17, 27, 42, 48, 71, 75]. The studies by Park et al. [68] and Yin et al. [87] investigate the bugs that need more than one fix attempt. Our study reuses the bugs used by Park et al. [68] because answering some of our research questions requires distinguishing between bugs that were fixed correctly on the first attempt and bugs that required several attempts to be fully fixed. However, the study by Park et al. is *not* related to performance bugs. Unlike all these studies of generic bugs, our study focuses on performance bugs.

Detecting Performance Bugs and Improving Performance: There is much work on detecting performance bugs and improving performance. Most of the work identifies code locations that take a long time to execute [18,22,31,57,93]. Several techniques [40,66,76] identify performance problems by detecting either anomalous or unexpected behavior. Other techniques [8, 25, 84, 86] detect runtime bloat, i.e., operations that perform a lot of work to accomplish simple tasks. Several techniques generate or select tests for performance testing [13, 29, 94]. All these techniques give good insight about some particular sources and causes of performance bugs. Unlike these specific techniques, our study analyzes more generally how performance bugs are discovered, reported, and fixed by developers, and is thus complementary.

5.2 Work Related to TODDLER and LULLABY

Profiling, Visualization, and Computational Complexity: Profiling and performance visualization tools are critical for developers to understand the performance features of different software components. A lot of recent progress was made to provide more accurate and efficient profiling [18,22–24,31,32,38,57,93]. However, as discussed in Section 1.3, profilers have fundamental limitations in detecting performance bugs. Several tools estimate the worst-time computational complexity of code [13, 20, 30], but like profilers, these tools report that some computation takes time, not if it wastes time. TODDLER complements these techniques to find performance bugs.

Performance-Bug Detection: Several techniques detect the excessive use of temporary objects, a common performance problem in object-oriented software [25,84]. Xu et al. use a run-time

analysis to detect low-utility data structures where the effort to construct member fields outweighs the usage of these fields [85]. Jin et al. study efficiency rules in performance-bug patches and detect performance bugs that are similar with previously patched ones [34]. Other techniques detect performance problems caused by idle time [3], multi-thread false sharing [47], or error recovery in distributed systems [40]. The success of these tools demonstrates the potential of performance-bug detection, but the existing work only covers a small portion of real-world performance bugs. TODDLER focuses on performance bugs caused by inefficient or redundant computation across nested loops. Many of these bugs, such as the real-world example bugs discussed in Chapter 3, cannot be detected by the existing performance bug detectors. Therefore, TODDLER complements these techniques well.

Automatic Bug Fixing: Several recent techniques have been propose to automatically fix bugs. GenProg [81] uses genetic programming to repair code. LASE [51], SysEdit [50], and FixWizard [61] repair code using similar edits from the fixes of previous bugs, SemFix [59] uses semantic analysis, PAR [41] learns from fixes written by developers, PACHIKA [21] analyzes differences in program behavior, and AutoFix-E [80] uses software contracts. PHPQuickFix and PHPRepair [72] and PhpSync [60] fix PHP code. Axis [46] and AFix [35] automatically fix atomicity violations. Unlike these techniques, LULLABY automatically fixes performance bugs. Furthermore, taking advantage of the unique properties of the bugs it detects, LULLABY successfully fixes 149 out of 150 bugs, with one bug not fixed due to a technical limitation in WALA.

Chapter 6

Conclusions and Future Work

Software performance is critical for how end-users perceive the quality of software products. While profilers are successful at localizing expensive computation, effectively preventing performance bugs from escaping to production requires a more comprehensive tool support. This dissertation makes three contributions towards improved handling of performance bugs: (1) understanding how performance bugs are discovered, reported to developers, and fixed by developers in comparison to non-performance bugs, (2) a novel automated oracle for performance bugs, which enables testing for performance bugs using the well established and automated process of testing for functional bugs, and (3) a novel technique for detecting and fixing performance bugs that have non-intrusive fixes and that are likely to be adopted by developers.

We next present several possible future projects building on our experience of working with performance bugs:

Performance Bug Fixing: During the LULLABY project, we found that developers greatly appreciate tools that can generate or suggest patches for performance bugs. This observation is also consistent with our findings in Chapter 2, and it is intuitive. LULLABY can generate patches only for a very restrictive set of performance bugs (which LULLABY also detects). The more generic problem is how to generate patches for generic performance bugs. Generic patch generation is a difficult problem [28], but we believe leveraging additional information that is specific to performance bugs may make the problem more tractable.

Test Generation for Performance Bugs: During the TODDLER project, one major problem we faced was how to write performance tests that can be analyzed by a dynamic technique. We plan to leverage our experience with generating efficient tests for multi-threaded code [63] and

generate tests that can expose performance bugs.

Hybrid Static-Dynamic Detection of Other Types of Performance Bugs: During our LULLABY and TODDLER projects we found that static and dynamic detection techniques for performance bugs each have its advantages and disadvantages. For example, dynamic techniques are precise but may miss bugs not triggered during the monitored runs, while static techniques are less precise but may cover the entire possible program behavior. We may consider techniques similar in spirit to prior work [37,65,73], but applied to performance bugs. Such techniques would likely use additional information specific to performance bugs. This information would likely be provided by an analysis similar to the analyses described in Chapter 3 and Chapter 4.

Framework for Writing Performance Tests and Assertions: The success of non-performance bug testing is in part due to the disciplined testing adopted by software projects and the tools that enable this disciplined testing. For example, for non-performance bugs, developers can easily write JUnit tests and assertions for the expected results. Similar attempts of disciplined testing have recently been made for testing concurrent code [14, 33]. Unfortunately, no technique exists that offers facilities similar to unit tests and assertions for performance bugs. We can use some of the experience we accumulated while writing tests for TODDLER and while inspecting tests written by developers to build a framework that makes it easy for developers to write performance tests and assertions.

References

[1] *JIP, the Java interactive profiler*, http://jiprof.sourceforge.net/.

[2] *WALA: T.J. Watson libraries for analysis*, http://wala.sourceforge.net.

[3] Erik A. Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell, *Performance analysis of idle programs*, OOPSLA, 2010.

[4] Apache Software Foundation, *Apache's JIRA issue tracker*, https://issues.apache.org/jira/secure/Dashboard.jspa.

[5] Apple Computer, Inc., *GProf manual pages*, http://www.manpagez.com/man/1/gprof/.

[6] Jorge Aranda and Gina Venolia, *The secret life of bugs: Going past the errors and omissions in software repositories*, ICSE, 2009.

[7] Muhammad Asaduzzaman, Michael C. Bullock, Chanchal K. Roy, and Kevin A. Schneider, *Bug introducing changes: A case study with Android*, MSR, 2012.

[8] Suparna Bhattacharya, Mangala Gowri Nanda, K. Gopinath, and Manish Gupta, *Reuse, recycle to de-bloat software*, ECOOP, 2011.

[9] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar T. Devanbu, *Fair and balanced?: Bias in bug-fix datasets*, FSE'09.

[10] Michael D. Bond and Kathryn S. McKinley, *Probabilistic calling context*, OOPSLA, 2007.

[11] Randal E. Bryant and David R. O'Hallaron, *Computer systems: A programmer's perspective*, Addison-Wesley, 2010.

[12] Bugzilla@Mozilla, *Bugzilla keyword descriptions*, https://bugzilla.mozilla.org/describekeywords.cgi.

[13] Jacob Burnim, Sudeep Juvekar, and Koushik Sen, *WISE: Automated test generation for worst-case complexity*, ICSE, 2009.

[14] Jacob Burnim and Koushik Sen, *Asserting and checking determinism for multithreaded programs*, ESEC/SIGSOFT FSE, 2009.

[15] Raymond P. L. Buse and Westley Weimer, *The road not taken: Estimating path execution frequency statically*, ICSE, 2009.

[16] Subhachandra Chandra and Peter M. Chen, *Whither generic recovery from application faults? A fault study using open-source software*, DSN, 2000.

[17] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler, *An empirical study of operating system errors*, SOSP, 2001.

[18] Emilio Coppa, Camil Demetrescu, and Irene Finocchi, *Input-sensitive profiling*, PLDI, 2012.

[19] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to algorithms*, The MIT Press, Cambridge, MA, 1990.

[20] Scott A. Crosby and Dan S. Wallach, *Denial of service via algorithmic complexity attacks*, USENIX Security Symposium, 2003.

[21] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer, *Generating fixes from object behavior anomalies*, ASE, 2009.

[22] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi, *Mining hot calling contexts in small space*, PLDI, 2011.

[23] John Demme and Simha Sethumadhavan, *Rapid identification of architectural bottlenecks via precise event counting*, ISCA, 2011.

[24] Amer Diwan, Matthias Hauswirth, Todd Mytkowicz, and Peter F. Sweeney, *TraceAnalyzer: A system for processing performance traces*, Softw., Pract. Exper. **41** (2011), no. 3, 267–282.

[25] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky, *A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications*, FSE, 2008.

[26] David Fields and Bill Karagounis, *Inside Windows 7—reliability, performance and PerfTrack*, 2009, http://channel9.msdn.com/Blogs/Charles/Inside-Windows-7-Reliability-Performance-and-PerfTrack.

[27] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues, *A study of the internal and external effects of concurrency bugs*, DSN, 2010.

[28] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer, *A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each*, ICSE, 2012.

[29] Mark Grechanik, Chen Fu, and Qing Xie, *Automatically finding performance problems with feedback-directed learning software testing*, ICSE'12.

[30] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi, *SPEED: Precise and efficient static estimation of program computational complexity*, POPL, 2009.

[31] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie, *Performance debugging in the large via mining millions of stack traces*, ICSE, 2012.

[32] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer, *Automating vertical profiling*, OOPSLA, 2005.

[33] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Rosu, and Darko Marinov, *Improved multithreaded unit testing*, FSE, 2011.

[34] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu, *Understanding and detecting real-world performance bugs*, PLDI, 2012.

[35] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit, *Automated atomicity-violation fixing*, PLDI, 2011.

[36] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu, *Automated concurrency-bug fixing*, OSDI, 2012.

[37] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen, *CalFuzzer: An extensible active testing framework for concurrent programs*, CAV, 2009.

[38] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth, *Catch me if you can: Performance bug detection in the wild*, OOPSLA, 2011.

[39] Paul Kallender, *Trend Micro will pay for PC repair costs*, 2005, http://www.pcworld.com/article/120612/article.html.

[40] Charles Edwin Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala, *Finding latent performance bugs in systems implementations*, FSE, 2010.

[41] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim, *Automatic patch generation learned from human-written patches*, ICSE, 2013.

[42] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park, *Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts*, TSE (2011).

[43] Donald E. Knuth, *Structured programming with go to statements*, ACM Comput. Surv. **6** (1974), no. 4, 261–301.

[44] Chris Lattner and Vikram Adve, *LLVM: A compilation framework for lifelong program analysis & transformation*, CGO, 2004.

[45] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz, *R2Fix: Automatically generating bug fixes from bug reports*, ICST'13.

[46] Peng Liu and Charles Zhang, *Axis: Automatically fixing atomicity violations through solving control constraints*, ICSE, 2012.

[47] Tongping Liu and Emery D. Berger, *Precise detection and automatic mitigation of false sharing*, OOPSLA, 2011.

[48] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou, *Learning from mistakes: A comprehensive study on real world concurrency bug characteristics*, ASPLOS, 2008.

[49] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood, *Pin: Building customized program analysis tools with dynamic instrumentation*, PLDI, 2005.

[50] Na Meng, Miryung Kim, and Kathryn S. McKinley, *Systematic editing: Generating program transformations from an example*, PLDI, 2011.

[51] _____, *LASE: Locating and applying systematic edits by learning from examples*, ICSE, 2013.

[52] Microsoft Corp., *Connect*, https://connect.microsoft.com/.

[53] _____, *How do I use the profiler in Windows Phone Mango?*, http://msdn.microsoft.com/en-us/windowsmobile/Video/hh335849.

[54] Domas Mituzas, *Embarrassment*, 2009, http://dom.as/2009/06/26/embarrassment/.

[55] Ian Molyneaux, *The art of application performance testing: Help for programmers and quality assurance*, O'Reilly Media, 2009.

[56] Glen Emerson Morris, *Lessons from the Colorado benefits management system disaster*, 2004, www.ad-mkt-review.com/public_html/air/ai200411.html.

[57] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney, *Evaluating the accuracy of Java profilers*, PLDI, 2010.

[58] Mayur Naik, Alex Aiken, and John Whaley, *Effective static race detection for Java*, PLDI, 2006.

[59] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra, *SemFix: Program repair via semantic analysis*, ICSE, 2013.

[60] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen, *Auto-locating and fix-propagating for HTML validation errors to PHP server-side code*, ASE, 2011.

[61] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen, *Recurring bug fixes in object-oriented programs*, ICSE, 2010.

[62] Adrian Nistor, Tian Jiang, and Lin Tan, *Discovering, reporting, and fixing performance bugs*, MSR, 2013.

[63] Adrian Nistor, Qingzhou Luo, Michael Pradel, Thomas R. Gross, and Darko Marinov, *Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code*, ICSE, 2012.

[64] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu, *Toddler: Detecting performance problems via similar memory-access patterns*, ICSE, 2013.

[65] Robert O'Callahan and Jong-Deok Choi, *Hybrid dynamic data race detection*, PPOPP, 2003, pp. 167–178.

[66] Jungju Oh, Christopher J. Hughes, Guru Venkataramani, and Milos Prvulovic, *LIME: A framework for debugging load imbalance in multi-threaded execution*, ICSE, 2011.

[67] Oracle Corporation, *HPROF JVM profiler*, http://java.sun.com/developer/technicalArticles/Programming/HPROF.html.

[68] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae, *An empirical study of supplementary bug fixes*, MSR, 2012.

[69] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang, *Automated support for classifying software failure reports*, ICSE, 2003.

[70] Tim Richardson, *1901 census site still down after six months*, 2002, http://www.theregister.co.uk/2002/07/03/1901_census_site_still_down/.

[71] Swarup Kumar Sahoo, John Criswell, and Vikram S. Adve, *An empirical study of reported bugs in server software with implications for automated bug diagnosis*, ICSE, 2010.

[72] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren, *Automated repair of HTML generation errors in PHP applications using string constraint solving*, ICSE, 2012.

[73] Koushik Sen, *Race directed random testing of concurrent programs*, PLDI, 2008.

[74] Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov, *Testing container classes: Random or systematic?*, FASE, 2011.

[75] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan, *High-impact defects: A study of breakage and surprise defects*, FSE, 2011.

[76] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake, *Predicting performance via automated feature-interaction detection*, ICSE, 2012.

[77] Soot, *Soot: A Java optimization framework*, http://www.sable.mcgill.ca/soot/.

[78] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield, *Analyzing lock contention in multithreaded applications*, PPOPP, 2010.

[79] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park, *Model checking programs*, ASE-J (2003).

[80] Yi Wei, Yu Pei, Carlo A. Furia, Lucas Serpa Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller, *Automated fixing of programs with contracts*, ISSTA, 2010.

[81] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest, *Automatically finding patches using genetic programming*, ICSE, 2009.

[82] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie, *Context-sensitive delta inference for identifying workload-dependent performance bottlenecks*, ISSTA, 2013.

[83] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma, *Ad hoc synchronization considered harmful*, OSDI, 2010.

[84] Guoqing (Harry) Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky, *Go with the flow: Profiling copies to find runtime bloat*, PLDI, 2009.

[85] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky, *Finding low-utility data structures*, PLDI, 2010.

[86] Dacong Yan, Guoqing Xu, and Atanas Rountev, *Uncovering performance problems in Java applications with reference propagation profiling*, ICSE, 2012.

[87] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram, *How do fixes become bugs?*, FSE, 2011.

[88] Shin Yoo and Mark Harman, *Regression testing minimization, selection and prioritization: A survey*, Softw. Test., Verif. Reliab. (2012).

[89] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi, *Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge*, ISSTA, 2009.

[90] Yourkit LLC, *Yourkit profiler*, http://www.yourkit.com.

[91] Shahed Zaman, Bram Adams, and Ahmed E. Hassan, *Security versus performance bugs: A case study on Firefox*, MSR, 2011.

[92] ———, *A qualitative study on performance bugs*, MSR, 2012.

[93] Dmitrijs Zaparanuks and Matthias Hauswirth, *Algorithmic profiling*, PLDI'12.

[94] Pingyu Zhang, Sebastian G. Elbaum, and Matthew B. Dwyer, *Automatic generation of load tests*, ASE, 2011.