Discovering, Reporting, and Fixing Performance Bugs

Adrian Nistor[1], Tian Jiang[2], and Lin Tan[2]
[1]University of Illinois at Urbana-Champaign, [2]University of Waterloo
nistor1@illinois.edu, {t2jiang, lintan}@uwaterloo.ca

***Abstract*—Software performance is critical for how users perceive the quality of software products. Performance bugs—programming errors that cause significant performance degradation—lead to poor user experience and low system throughput. Designing effective techniques to address performance bugs requires a deep understanding of how performance bugs are discovered, reported, and fixed.**

In this paper, we study how performance bugs are discovered, reported to developers, and fixed by developers, and compare the results with those for non-performance bugs. We study 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.

I. Introduction

Software performance is important to the overall success of a software project. Performance bugs—programming errors that create significant performance degradation [1]—hurt software performance and quality. They lead to poor user experience, degrade application responsiveness, lower system throughput, and waste computational resources [2], [3]. Even expert programmers introduce performance bugs, which have already caused serious problems [4]–[7]. Well tested commercial products such as Internet Explorer, Microsoft SQLServer, and Visual Studio are also affected by performance bugs [8].

Therefore, 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 [9], and operating systems such as Windows 7 provide built-in support for tracking operating system performance [10]. In addition, many techniques are proposed recently to detect various types of performance bugs [11]–[23].

To understand the effectiveness of these techniques and design new effective techniques for addressing performance bugs requires a deep understanding of performance bugs. A few recent papers [24]–[26] 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" [27], 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
- 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 is qualified as a performance bug.

  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]–[31], 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 [32]–[34]? 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.

978-1-4673-2936-1/13/$31.00 © 2013 IEEE

MSR 2013, San Francisco, CA, USA

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 and any similar study (details in Section II-A). 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.

This paper 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 [35], [36], 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 [36], 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. [35].

  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 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 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]–[31], 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.

II. Experimental Methods

A. 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. [35], 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 paper. 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 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. [35], 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 I summarizes the characteristics of the studied subjects: time period between the first and last commits in the

TABLE I
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

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 I give the number of performance and non-performance bugs studied in this paper. 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 II-B 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 IV). 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 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 II-B and discuss the threats to this method in Section IV.

B. 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 [24], [25], [37], [38]. We first identify the bug reports that contains 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 authors. For the bugs where the results from the two inspections differ, the authors 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.

C. 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 paper, 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 [24], [25], [37], [38]), 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 [35], [37], [39], the studied time period and projects are not randomly selected. We discuss these threats further in Section IV.

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.

III. Results

A. 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. [35] 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 II 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 II 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 II
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 III 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***.

TABLE III
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

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 patch; Section III-B 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 IV 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.

TABLE IV
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

Bug and Patch Examples: Figure 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, and implements this better hash method in the second patch, as shown in Figure 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 ...
```

Fig. 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 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. Among so many changes, the developer got one pattern wrong (line 5 in Figure 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 ...
```

Fig. 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.

B. 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 V 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 V
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 VI 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 II-C (denoted by "/").

TABLE VI
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 II-C.

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

Figure 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 III-A), the performance bugs that do need supplementary patches are more difficult to fix.

Figure 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 V). 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 VII 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

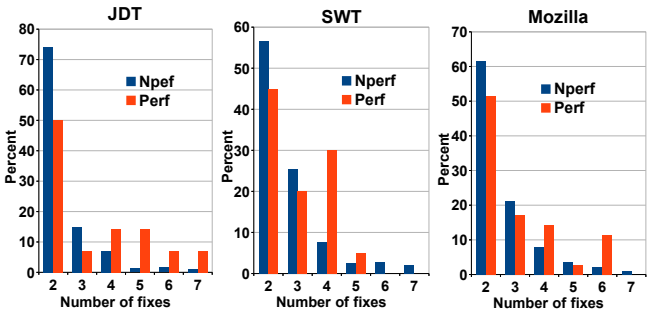

Fig. 3. Number of times a bug is fixed. This figure uses the multi-patch bugs from I-Perf and A-NonPerf.

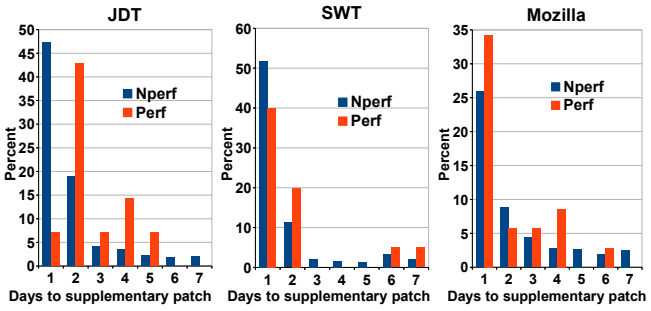

Fig. 4. Time (number of days) between the first patch and the last supplementary patch. This figure uses the I-Perf and A-NonPerf datasets.

average number 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 VII
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

Table VIII 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 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 III-A), when they do, the supplementary patches are much more complex than for non-performance bugs.

TABLE VIII
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 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.

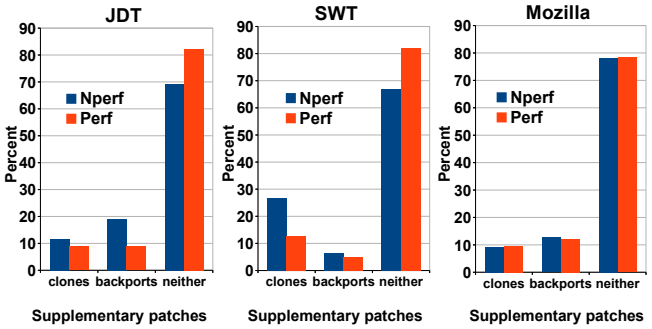

Fig. 5. Supplementary patches that are clones, backports, or none of the two. This table uses the I-Perf and A-NonPerf datasets.

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 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.

C. 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 IX 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 IX
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 X 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.

TABLE X
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

What additional information is provided with the bug report is not necessarily dependent on how the bugs were discovered (Table IX). 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 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.

IV. 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. [26] 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 [24], [37], [38].

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 [35], 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 [40] and can be incomplete or incorrect [41]. 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. [35] 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 I), 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.

V. Related Work

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. [25] 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 [26] 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 X, 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. [24] 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., [37]–[39], [41]–[46]. The studies by Park et al. [35] and Yin et al. [36] investigate the bugs that need more than one fix attempt. Our study reuses the bugs used by Park et al. [35] 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 [11]–[13], [34], [47]. Several techniques [14]–[16] identify performance problems by detecting either anomalous or unexpected behavior. Other techniques [17]–[20] 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 [21]–[23]. 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.

VI. Conclusions

Performance bugs create problems even for well tested software written by expert programmers [4]–[8]. This paper 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.

Acknowledgments

We thank Jihun Park and Miryung Kim for providing the data from their study, Mitchell Jameson for helping with experiments, and Darko Marinov for his valuable discussion and feedback. We also thank the University of Waterloo's statistical counseling service, William Marshall, and Mladen Laudanovic for help with the statistical analysis. This work is partially supported by the National Science and Engineering Research Council of Canada, a Google gift grant, and the US National Science Foundation under Grant No. CCF-0746856.

References

[1] Bugzilla@Mozilla, "Bugzilla keyword descriptions," https://bugzilla.mozilla.org/describekeywords.cgi.
[2] I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.
[3] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Addison-Wesley, 2010.
[4] P. Kallender, "Trend Micro will pay for PC repair costs," 2005, http://www.pcworld.com/article/120612/article.html.
[5] G. E. Morris, "Lessons from the Colorado benefits management system disaster," 2004, www.ad-mkt-review.com/public_html/air/ai200411.html.
[6] T. Richardson, "1901 census site still down after six months," 2002, http://www.theregister.co.uk/2002/07/03/1901_census_site_still_down/.
[7] D. Mituzas, "Embarrassment," 2009, http://dom.as/2009/06/26/embarrassment/.
[8] Microsoft Corp., "Connect," https://connect.microsoft.com/.
[9] Apache Software Foundation, "Apache's JIRA issue tracker," https://issues.apache.org/jira/secure/Dashboard.jspa.
[10] D. Fields and B. Karagounis, "Inside Windows 7—reliability, performance and PerfTrack," 2009, http://channel9.msdn.com/Blogs/Charles/Inside-Windows-7-Reliability-Performance-and-PerfTrack.
[11] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *ICSE*, 2012.
[12] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," in *PLDI*, 2012.
[13] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," in *PLDI'12*.
[14] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic, "LIME: A framework for debugging load imbalance in multi-threaded execution," in *ICSE*, 2011.
[15] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *ICSE*, 2012.
[16] C. E. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala, "Finding latent performance bugs in systems implementations," in *FSE*, 2010.
[17] G. H. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, "Go with the flow: Profiling copies to find runtime bloat," in *PLDI*, 2009.
[18] D. Yan, G. Xu, and A. Rountev, "Uncovering performance problems in Java applications with reference propagation profiling," in *ICSE*, 2012.
[19] B. Dufour, B. G. Ryder, and G. Sevitsky, "A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications," in *FSE*, 2008.
[20] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta, "Reuse, recycle to de-bloat software," in *ECOOP*, 2011.
[21] P. Zhang, S. G. Elbaum, and M. B. Dwyer, "Automatic generation of load tests," in *ASE*, 2011.
[22] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *ICSE'12*.
[23] J. Burnim, S. Juvekar, and K. Sen, "WISE: Automated test generation for worst-case complexity," in *ICSE*, 2009.
[24] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012.
[25] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: A case study on Firefox," in *MSR*, 2011.
[26] ——, "A qualitative study on performance bugs," in *MSR*, 2012.
[27] D. E. Knuth, "Structured programming with go to statements," *ACM Comput. Surv.*, vol. 6, no. 4, pp. 261–301, Dec. 1974.
[28] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *PLDI*, 2011.
[29] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012.
[30] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *OSDI*, 2012.
[31] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2Fix: Automatically generating bug fixes from bug reports," in *ICST'13*.
[32] Sun Microsystems, "HPROF JVM profiler," http://java.sun.com/developer/technicalArticles/Programming/HPROF.html.
[33] Yourkit LLC, "Yourkit profiler," http://www.yourkit.com.
[34] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers," in *PLDI*, 2010.
[35] J. Park, M. Kim, B. Ray, and D.-H. Bae, "An empirical study of supplementary bug fixes," in *MSR*, 2012.
[36] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram, "How do fixes become bugs?" in *FSE*, 2011.
[37] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *DSN*, 2010.
[38] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008.
[39] S. K. Sahoo, J. Criswell, and V. S. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *ICSE*, 2010.
[40] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *FSE'09*.
[41] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *ICSE*, 2009.
[42] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? A fault study using open-source software," in *DSN*, 2000.
[43] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," in *SOSP*, 2001.
[44] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: A study of breakage and surprise defects," in *FSE*, 2011.
[45] M. Asaduzzaman, M. C. Bullock, C. K. Roy, and K. A. Schneider, "Bug introducing changes: A case study with Android," in *MSR*, 2012.
[46] D. Kim, X. Wang, S. Kim, A. Zeller, S.-C. Cheung, and S. Park, "Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *TSE*, 2011.
[47] D. C. D'Elia, C. Demetrescu, and I. Finocchi, "Mining hot calling contexts in small space," in *PLDI*, 2011.