Empirical Studies of Performance Bugs and Performance Analysis Approaches for Software Systems

by

Shahed Zaman

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
April 2012

Abstract

Developing high quality software is of eminent importance to keep the existing customers satisfied and to remain competitive. One of the most important software quality characteristics is performance, which defines how fast and/or efficiently a software can perform its operation.

While several studies have shown that field problems are often due to performance issues instead of feature bugs, prior research typically treats all bugs as similar when studying various aspects of software quality (e.g., predicting the time to fix a bug) or focused on other types of bug (e.g., security bugs). There is little work that studies performance bugs.

In this thesis, we perform an empirical study to quantitatively and qualitatively examine performance bugs in the Mozilla Firefox and Google Chrome web browser projects in order to find out if performance bugs are really different from other bugs in practice and to understand the rationale behind those differences.

In our quantitative study, we find that performance bugs of the Firefox project take longer time to fix, are fixed by more experienced developers, and require changes to more lines of code. We also study performance bugs relative to security bugs, since security bugs have been extensively studied separately in the past. We find that security bugs are re-opened and tossed more often, are fixed and triaged faster,

are fixed by more experienced developers, and are assigned more number of developers in the Firefox project. Google Chrome project also shows different quantitative characteristics between performance and non-performance bugs and from the Firefox project.

Based on our quantitative results, we look at that data from a qualitative point of view. As one of our most interesting observation, we find that end-users are often frustrated with performance problems and often threaten to switch to competing software products.

To better understand, the rationale for some users being very frustrated (even threatening to switch product) even though most systems are well tested, we performed an additional study. In this final study, we explore a global perspective vs a user centric perspective of analyzing performance data. We find that a user-centric perspective might lead to a small number of users with considerably poor performance while the global perspective might show good or same performance across releases.

The results of our studies show that performance bugs are different and should be studied separately in large scale software systems to improve the quality assurance processes related to software performance.

Co-authorship

Earlier versions of the work in this thesis were published as listed below:

1. *Security versus performance bugs: a case study on Firefox (Chapter 3)*

   Shahed Zaman, Bram Adams, and Ahmed E. Hassan. In Proceedings of the 8th Working Conference on Mining Software Repositories (MSR), pages 93-102, Waikiki, Honolulu, HI, USA, 2011. ACM Press. (Acceptance ratio: 20/61 = 32.8%).

   My contribution - Drafting the research plan, expanding upon an existing collection of gathered data, analyzing the data, writing the manuscript and presenting the paper.

2. *A Qualitative Study on Performance Bugs (Chapter 4)*

   Shahed Zaman, Bram Adams, and Ahmed E. Hassan. To appear in the 9th Working Conference on Mining Software Repositories (MSR), Zürich, Switzerland, 2012. ACM Press. (Acceptance ratio: 18/64 = 28.13%).

   My contribution - Drafting the research plan, gathering and analyzing the data, and writing the manuscript.

3. *A Large Scale Empirical Study on User-Centric Performance Analysis (Chapter 5)*

Shahed Zaman, Bram Adams, and Ahmed E. Hassan. In Proceedings of the Industrial Track of the 5th International Conference on Software Testing, Verification and Validation (ICST) , Montréal, Québec, Canada. IEEE Computer Society Press. (Acceptance ratio: 9/33 = 30.30%).

My contribution - Drafting the research plan, gathering and analyzing the data, writing the manuscript and presenting the paper.

Acknowledgments

First and foremost, I would like to thank my supervisor Dr. Ahmed E. Hassan for his continuous support, great suggestion, advise and constant motivation throughout my thesis. I owe my deepest gratitude to Dr. Bram Adams for his enthusiasm, dedication, and patience. I have had a great research experience throughout this thesis with Ahmed and Bram. I would also like to show my gratitude to Dr. Mohammad Zulkernine for his suggestions, encouragement and help since I moved to Kingston.

I am also grateful to all my colleagues, at the Software Analysis and Intelligence Lab (SAIL) who have extended their help and encouraged me from time to time. I am very fortunate to work with the wonderful and thoughtful members of SAIL. I also thank the Relay Core Testing group at Research In Motion (RIM) for providing me the opportunity to conduct research on problems that are of high impact and importance to practice.

Finally, I would like to dedicate this work to my parents, wife and my brother. Without their support, encouragement and prayers, this thesis would not have been possible.

Statement of Originality

I, Shahed Zaman, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software performance is one of the most influential nonfunctional requirements [8], with the power to make or break a software system in today's competitive market. Software performance basically measures how fast and efficiently a software system can complete its operations [49]. Resolving performance related problems or performance bugs (e.g., high resource utilization or slow response time) is as important as adding new features to the system to keep the users satisfied and loyal to the software system. Studies show that many of the field problems reported are related to performance issues instead of feature bugs [42, 76].

Despite the importance of software performance, to the best of our knowledge, no studies have focused on performance bugs. One common theme in software quality assurance research is that most of them treat all bugs equally, i.e., they do not distinguish between different kinds of bugs [16, 19, 26, 33, 56, 58, 69, 81]. Security bugs have been studied extensively, while there are other types like usability, functionality, serviceability, performance.

Without a good understanding of the differences of performance bugs and their fixing process relative to other type of bugs, development resources cannot be properly

allocated and software releases may have critical performance issues. For instance, the Mozilla Firefox and Google Chrome web browser projects get a large amount of performance bugs reported due to browser plug-ins or extensions developed outside the project [78]. The decision of whether or not this type of performance bug will be tracked by the browser team, and how to do that, is crucial for the overall software quality assurance activity of the project, and requires in-depth research.

To better understand the differences between performance and non-performance bugs, and identify concrete ways to improve the performance bug resolution process, we need to empirically study software performance problems dealt within large software projects. Such a study will help stakeholders to better estimate software maintenance effort. For instance, (1) developers and testers could more accurately identify and understand performance problems and estimate the effort and actions required to fix them, and (2) project managers could have an idea of required resources and can develop performance maintenance policies.

Hence, this thesis qualitatively and quantitatively studies performance and non-performance bugs in the Mozilla Firefox and Google Chrome web browser projects. We find that performance bugs are different in terms of the time required to fix them, the experience of the developers who fix them, and the lines of code needed to fix them. Moreover, we find that performance bugs are harder to reproduce, require more collaborative effort to investigate and depend more on other bugs. Although, these findings are not always consistent between the studied projects, our findings show major difference in the characteristics of performance and non-performance bugs across and within projects.

As one of our findings, we find that despite consistent efforts for improving software

quality, some users complain (even threaten to switch a product) about software performance problems. This finding motivate us to study the benefit of user-centric performance analysis in practice, since system performance typically is evaluated from a global perspective where the efficiency (i.e., timing, resource utilization) is optimized across all users together, instead of being tailored to each (group of) user. Our analysis of test data from the performance test runs on two enterprise and one open source software systems show that customer-centric performance analysis can improve the performance analysis process by detecting performance deviations and trends that could not be detected from global perspective.

Based on our findings, we conjecture that performance problems should be explored in more detail with the software maintenance research in order to improve the maintenance process related to these important and critical bugs.

1.1 Research Statement

Prior research, experience analyzing large software projects, and interaction with industrial software performance testers (as part of a four month industrial embedding with an industrial performance engineering team) lead us to the following research hypothesis:

Performance bugs have different characteristics than other bugs and should be treated differently in software maintenance research and practice.

Motivation - Performance is considered to be an important software quality feature [52, 66]. However, performance bugs have not been studied/considered separately in

prior research on software maintenance. Intuitive reasoning for not considering performance bugs (despite knowing the importance of performance) in software maintenance research can be, performance bugs are of no difference than any other bug and/or performance bugs cannot be leveraged to improve the software performance and the overall quality of a software. We are interested in empirically studying performance bugs from large scale software systems. We want to study if performance bugs really have any different characteristics than other bugs, and if our studies on performance bugs can be used toward improving the processes related to software quality assurance.

1.2 Thesis Overview

We perform three different studies to explore our research statement:

1. *A quantitative study of performance bugs (Chapter 3)*

   We study the quantitative differences between performance, security and other bugs for the Mozilla Firefox and Google Chrome web browser projects. We choose the Firefox and Chrome projects from the same domain to see if our study findings are consistent between projects. These open-source projects have thousands of publicly available bug reports that we could easily extract and use in our study. Moreover, bug reports from the Mozilla project are commonly used for empirical research of bugs.

   We study security bugs along with performance bugs to provide context for our results since security bugs are commonly studied in literature due to their perceived importance and critical nature. We compare the studied bugs along three dimensions, i.e., Time, People and Bug Fix.

2. *A qualitative study of performance bugs (Chapter 4)*

   We study a random yet representative sample of performance and non-performance bug reports (and their comments) in the Mozilla Firefox and Google Chrome web browser projects. We take 100 random representative bug report samples of each type and for each project (400 bug reports in total), and examine the qualitative differences between performance and non-performance bugs. By qualitative differences, we refer to the different characteristics of bug reports related to human behavior and reasons that govern such behavior, i.e., the bug's impact on the user, context of the bug, the fix, and fix verification process. From the findings, we also try to find the rationale behind some of the differences observed in our quantitative study.

3. *A user-centric analysis of performance (Chapter 5)*

   To study that treating performance bugs differently can be beneficial, we dig more into one of the findings of our study on performance bugs. In Chapter 4, we find that users often complain (and threaten to switch) because of performance issues more often than non-performance issues. Inspired from this finding, we suggest the use of user-centric analysis for software performance.

   In user-centric analysis, software performance data is aggregated and analyzed for each (group of) user. As detailed data from the performance test runs on the Mozilla Firefox and Google Chrome projects are not available, we could not do this study on the same projects as our previous studies. We studied the performance test data of two enterprise system available from an industrial collaborator (Research In Motion) and one open-source performance benchmark system (Dell DVD Store) setup in our lab environment. In this study,

we compare the results from a user-centric analysis with the results from the commonly-used scenario-centric analysis, where performance data is aggregated for each scenario.

1.3 Major Thesis Contributions

In this thesis, we empirically study performance bugs and introduce a new approach for software performance analysis. In particular, our contributions are as follows:

1. To the best of our knowledge, we report the first ever quantitative study of performance bugs.

2. We perform the first ever qualitative study on performance bugs. We develop a taxonomy for the qualitative analysis of bug report data. We identify several sub-dimensions (tags) and group them into four dimensions namely, impact on the stakeholder, context of the bug, the bug fix, and the fix verification process. This taxonomy will help shape future qualitative studies on bug reports.

3. We propose a new approach (user-centric analysis) to analyze the performance of software systems. User-centric analysis approach can be used along with other existing approaches to provide a more complete view of the performance of a system.

1.4 Organization of the Thesis

The remainder of the thesis is organized as follows: Chapter 2 defines software performance, and performance bugs, and provides a brief background on software performance related research within the context of our thesis.

Chapter 3 represents the quantitative study that we performed on Mozilla Firefox and Google Chrome bug repository data. In Chapter 4, we describe our fine-grained qualitative study on performance and non-performance bugs. In Chapter 5, we dig more into software performance analysis based on one of our findings in qualitative study. In this chapter, we study the performance load test data of three software systems to empirically find out if customer-centric performance analysis is any useful and different than scenario-centric performance analysis.

Finally, Chapter 6 concludes the thesis, and discusses the limitations and potential future directions in this important area of software quality assurance.

Chapter 2

Background and Related Work

2.1 What is a performance bug?

For the purpose of this thesis, we define software performance bugs as follows:

A performance bug is a bug (i.e., problem, possible improvement, expectation) reported to a bug tracking system (also known as issue tracking system) that is related to software performance. Software performance refers to how fast and efficiently (resource utilization-wise) the system can perform its operation.

Performance bugs can be reported due to different reasons:

1. Performance regression

   When the software performance degrades compared to a prior release, a performance bug is reported and this phenomenon is called a performance regression.

2. Performance improvement suggestions

A developer, tester or an end-user may feel that the efficiency and/or the speed of a software operation could be improved and report that possible improvement as an issue to the issue tracking system.

3. Other performance problems

   A performance bug may be reported by any stakeholder (i.e., developer, manager, tester, end-user) when she perceives a performance problem that may effect the normal operation of the software, such as slow speed or poor resource utilization (i.e., high memory/CPU/disk usage).

2.2 Related Work

There has been a large amount of research on the detection of performance bugs. Jovian et al. worked on finding performance bugs by automatically monitoring the behavior of applications in order to provide helpful information for performance problem tracing to the developer [44]. Narpurkar et al. proposed efficient remote profiling of mobile devices to help users dynamically provide information to developers for isolating performance bugs. Prior work focuses on improving the detection and tracing processes for performance bugs, while we study the bug life cycle followed in practice.

We divide the rest of the related work into three subsections according to the three chapters in our study.

2.2.1 Bug classification (Chapter 3)

In Chapter 3, we classify bugs reported in issue tracking systems into three groups; performance, security and others. Here, we discuss related work in the areas of bug classification.

The literature contains several bug classification taxonomies [61]. The draft of the IEEE Standard Classification for Software Anomalies is a standard classification of software defects [1]. In this classification, software security and performance represent two out of six types of problems based on the effect of defects. Ahsan et al. propose an automatic bug triaging system based on a categorization of bug reports using text mining [5]. Gegick et al identify security bug reports via text mining [31].

2.2.2 Qualitative Analysis of Bug Repositories (Chapter 4)

In Chapter 4, we qualitatively study performance and non-performance bugs to get an understanding of human behaviors (reasoning) related to performance bugs. Here, we discuss related work in the areas of qualitative analysis of bug repositories.

Bertram et al. did a qualitative study to show that issue tracking systems are used as a communication and collaboration medium among customers, project managers, quality assurance personnel, and programmers [11]. Using questionnaire and semi-structured interviews, they found that issue trackers are an ever-growing knowledge store where each stakeholder contributes knowledge and information via individual bugs and features. In our study, we try to use this knowledge to learn about software performance bugs and their fixing process.

Zibran et al. qualitatively studied bug reports from different open-source projects to identify and rank usability issues related to designing and developing software APIs [80]. Guo et al. qualitatively studied bug reports from the Microsoft Windows Vista operating system project to find out the primary reasons for bug reassignment [35]. Guo et al. used this qualitative study on bug reports to provide support for their survey findings on Microsoft employees. Andrew et al. did a qualitative study on 100

bug reports from 3 different open-source software projects to find out how teams of distributed developers discuss and reach consensus in bug reports [47].

Bettenburg et al. performed a survey on developers and reporters related to software issue repositories in order to find out the qualities of a good bug report [13] [12]. In their survey, the developers and reporters identified and ranked different types of information provided in bug reports i.e., steps to reproduce, test cases, and stack traces. Our study also find active use of these type of information in bug reports (as found by Bettenburg et al.) to fix performance bugs.

2.2.3 User-centric analysis in software engineering (Chapter 5)

In Chapter 5, we propose the use of user-centric analysis in software performance analysis. Here, we discuss related work in the areas of software engineering and user-centric analysis. A user oriented approach has been considered before for different characteristics of software quality. In the field of software reliability modeling, Cheung described the software reliability from the point of view of a user [22] as "the probability that the program will give the desired output with a typical set of input data from that user environment". He showed that the reliability model can be formulated depending on the user profile (frequency distribution of the use of different features in the system).

User-centric analysis has also been considered in the field of usability analysis. Terry et al. used end-user oriented analysis on instrumented software systems to analyze the real-world practices of the users of that software [72]. They collected and analyzed different types of usage data to measure the usability of a software system. Calongne worked on web site and other usability goals [18]. One of those goals for

a designer is "high task performance", i.e., the quantifiable speed at which the web page should load and display the requested information given a particular system hardware and software configuration.

The ISO 9126 standard for software product quality has six characteristics that describe product quality [39]. "Efficiency" is the closest characteristic to the definition of performance [10]. This standard describes three set of metrics for software efficiency evaluation, i.e., time behavior metrics, resource utilization metrics and efficiency compliance metrics. ISO 9126 also mentions the view of software quality from three perspectives, user, developer and manager. Our empirical study results show the importance of considering the user's perspective for the analysis of these performance related metrics.

Chulani et al. derived a software quality view from customer satisfaction and service data to obtain a better understanding of the customer view of software quality [24]. Mockus et al. worked on finding the predictors of customer-perceived software quality [55]. From their study, they identified factors like software bug reports by customers, requests for assistance and field technician dispatches as the predictor of customer perceived quality for a large telecommunications software system. Mockus et al. also worked on relating the customer-perceived quality to process quality [54]. They developed and evaluated a practical measure of customer perceived quality based on the probability that a customer will observe a problem. The measure is calculated from software problem tracking and customer support systems data. Our analysis shows the importance of such user-centric analysis for a sub-characteristic of software quality, i.e., software performance.

Gould et al. theoretically and empirically recommended three principles of system

design that should be followed to design a useful and easy to use software system [32]. The three recommended principles were, to focus on (1) users and task early in the development process, (2) empirical measurement, and (3) iterative design. They also emphasized the importance of designer understanding the users of the system by studying the users directly (by observing their cognitive, behavioral characteristic) or in part (by observing the nature of the work expected to be accomplished) from the early stages of the development life cycle. In our study, we also find that considering the user's perspective for software performance analysis is beneficial for detecting performance issues that might go unnoticed.

2.3 Chapter Summary

In this chapter, we define software performance and performance bugs in the context of our study. We also survey prior work focused on four topics: performance bugs, bug classification, qualitative analysis on bug repositories, and user-centric analysis in software engineering. In the next chapter, we quantitatively study the difference among performance, security and other bugs in the Mozilla Firefox (primarily) and Google Chrome project.

Chapter 3

A Quantitative Study of Performance Bugs

Chapter Overview: *A good understanding of the impact of different types of bugs on various project aspects is essential to improve software quality research and practice. For instance, we would expect that security bugs are fixed faster than other types of bugs due to their critical nature. However, prior research has often treated all bugs as similar when studying various aspects of software quality (e.g., predicting the time to fix a bug), or has focused on one particular type of bug (e.g., security bugs) with little comparison to other types. In this chapter, we study how different types of bugs (performance and security bugs) differ from each other and from the rest of the bugs in a software project. Through a case study on the Firefox project, we find that security bugs are fixed and triaged much faster, but are reopened and tossed more frequently. Furthermore, we also find that security bugs involve more developers and impact more files in a project. This study is the first work to ever empirically study performance bugs and compare it to frequently-studied security bugs. Our findings highlight the importance of considering the different types of bugs in software quality research and practice.*

Previous studies show that maintenance and evolution activities represent over 90% of the software development cost [30, 69]. Improving quality assurance effort in software projects is an important task to keep the existing customers satisfied and to compete in a competitive market. Research has focused on improving the quality assurance of software projects. For example, software defect prediction models use

various code, process, social structure, geographical distribution and organizational structure metrics to predict the number of defects and their locations [16, 19, 26, 33, 56, 58, 69, 81]. Other work focuses on predicting the time it takes to fix a bug [46, 62, 75], and which developer should fix a bug [6].

One common theme in the aforementioned quality assurance research is that most of them treat all bugs equally, i.e., they do not distinguish between different kinds of bugs. For example, one would expect bugs that are labeled as security risks to experience a different treatment than typos in code comments or user interface quirks. Yet, most techniques build generic models for generic bugs.

In order to establish our research hypothesis that performance bugs should be treated differently in software maintenance research and practice, we need to have some evidence that performance bugs are really different by nature. Moreover, we believe that a thorough understanding of the various aspects associated with the different types of bugs is needed. In this chapter, we quantitatively study different characteristics for security, performance and other bugs in the Firefox open source project to show that these characteristics differ between different bug types in practice. We study security and performance bugs, since they are important non-functional types of bugs. Security bugs are of high risk, and performance issues occur commonly in the field [41]. Yet, both types of bug have never been compared against each other to see if there are differences in their bug life cycle characteristics and if these differences are likely to impact or affect current work in the area of software quality assurance.

We address the following three research questions:

Q1). How fast are bugs fixed?

On average, security bugs are fixed 2.8 times faster than performance bugs, but

they are reopened 2.5 times more than performance bugs and approximately 4.5 times more than other bugs.

On average, security bugs are triaged 3.64 times faster than performance bugs and 3.38 times faster than the rest of the bugs, but security bugs are tossed 2.67 times more than performance and 4.7 times more than the rest of the bugs.

Q2). Who fixes bugs?

On average, security bugs are assigned 2.39 times more developers than performance bugs are, and 3.51 times more developers than the rest of the bugs. Performance and security bugs are fixed by more experienced developers.

Q3). What are the characteristics of bug fixes?

On average, security bug fixes are 1.48 times more complex than performance bug fixes and 1.6 times more complex than the fixes for other bugs.

On average, fixing a performance bug requires change in 2.6 times more files than security bugs.

Overview of the chapter: Section 3.1 discusses our approach. Section 3.2 is divided into three sub-sections according to the studied research questions. Each subsection of Section 3.2 discusses a motivation, approach and the results of the specific question. Finally, Section 3.4 discusses threats to validity and Section 3.5 presents the conclusion of the chapter related to our research hypothesis.

3.0.1 Choice of Subject Systems

In our case study, we primarily studied the Firefox web browser, since Firefox is one of the most popular web browsers and runs on different platforms and system

environments. For a complex software like Firefox, performance and security are two major software quality requirements that matters. Moreover, the issue tracking system used by Mozilla (Bugzilla), contains and documents all reported bugs that are available online. For our study, we started from Mozilla's Bugzilla data from September, 1994 to August 15, 2010.

First, we had to identify the bugs that are related to the Firefox web browser, since Mozilla's bug tracking system manages multiple projects, including Firefox. Bugzilla contains 567,595 bug reports of different Mozilla components, like Core, Firefox, SeaMonkey, and Thunderbird. For Firefox, we took bug reports that are related to Core (shared components used by Firefox and other Mozilla software, including the handling of Web content) and to Firefox.

3.1 Study Design

This section presents the design of our case study to compare security, performance and other bugs in Firefox. Figure 3.1 shows an overview of our approach. First, we extract the necessary data from the bug repository (Bugzilla) and source code repository (CVS). Then, we classify the bug reports related to performance and security. Using CVS data, we also identify the bug fix information. For every type of bug, we calculate several metrics, then statistically compare the metrics across the types of bugs (performance, security and other). This section elaborates on each of these steps.

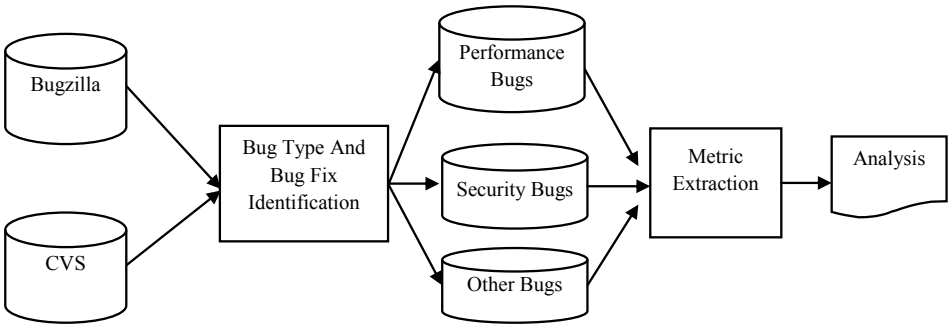

Figure 3.1: Overview of our approach to study the differences in characteristics of security, performance and other bugs.

3.1.1 Bug Type Classification

Bugzilla does not have a built-in categorization or tagging for performance and security bugs. The Firefox project team uses the keyword 'perf' for performance-related bugs, but this tagging is not mandatory and we found many bugs that do not have any such tag.

Hence, to classify performance bugs, we had to use heuristics. We looked for the keywords 'perf', 'slow', and 'hang' in the bug report title and keyword field. Although the keyword 'perf' gave us bug reports that had 'performance' in their title, we found that there are many unrelated bug reports containing the word 'perfect', 'performing' or 'performed'. We had to automatically exclude these keywords from the list using regular expressions. Using these heuristics, we found 7,603 performance bugs.

Since heuristics are not perfect, we performed a statistical sampling to estimate the precision and recall of our heuristics. To calculate precision with a 95% confidence level and a confidence interval of 10, we randomly sampled and checked 95 of the 7,603 bugs classified as performance-related [45]. All 95 randomly selected samples were

Table 3.1: Confusion matrix for Firefox dataset

Actual →	Firefox	
Prediction ↓	**P**	**NP**
P	95	0
NP	19	77

indeed performance bugs, yielding a precision of 100 ± 10%. To calculate recall, we did statistical sampling on the 287,595 bugs classified as non-performance. For the same 95% confidence level and confidence interval of 10, we randomly sampled and checked 96 bugs. 19 out of these 96 sampled bug reports were found to be related to performance, yielding a recall of 80 ± 10% (see confusion matrix in Table 3.1).

To identify the security bugs, we use the Mozilla Foundation Security Advisory (MFSA) [2]. MFSA contains links to security bugs for every issued advisory. MFSA has been issuing security advisories for Mozilla's products since 2005. Based on the MFSA, we extracted a dataset of 847 security and 294,351 non-security bugs. As security bugs were marked by the experts of the security advisory team of Firefox, we did not need to estimate the precision and recall as we did for performance bugs.

3.1.2 Distribution of Bugs

Figure 3.2 shows the number of reported bugs over time in log scale. The earliest security bug in our data was reported in May, 2003. 2003 was also the year in which the Firefox project started. Hence, we did not study bugs reported prior to May, 2003. This left us with 4,293 performance bugs, 847 security bugs and 178,531 other bugs for our study (May, 2003-August, 2010). Performance and security bugs have 11 bugs in common that had a security threat and caused the browser to hang/crash.

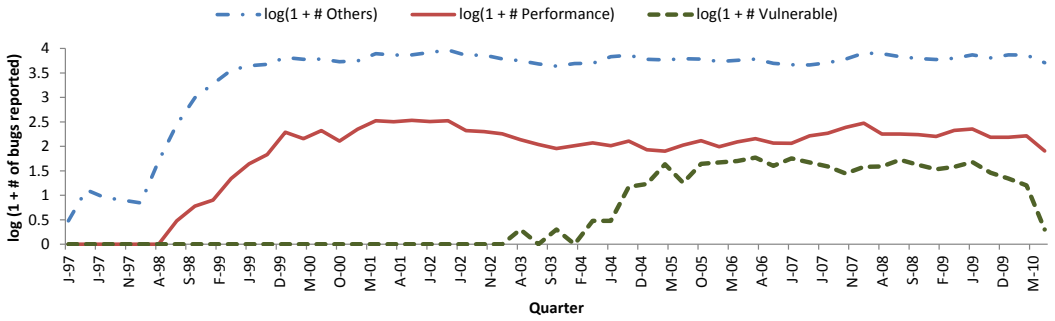

Figure 3.2: Distribution of the number of bugs reported (quarterly) for security, performance and other bugs.

We put these 11 bugs in both the security and performance groups of our study.

To better understand the composition of each group of bugs, we used a topic model [17]. Topics are collections of words that co-occur frequently in a corpus of text. These topics describe the major themes that span a corpus [74], which provides a means of automatically summarizing and organizing the various data of a software project. Recently, researchers have applied topic models to various aspects of a software project, including source code [7, 48, 50, 64] and documentation [27, 37]. Statistical topic models, such as latent Dirichlet allocation (LDA) [17], can automatically discover a set of topics within a corpus.

We ran LDA on the bug report comments to find out the topics that describe the major themes that span these performance and security bugs. From the LDA output (results are in Appendix A) of performance bug comments, we found two major subcategories of performance bugs: (1) Plugin-related (like java applet plugin and the flash plugin) and (2) GUI-related (user interface related libraries, like gfx and gecko) performance issues. Other smaller types of performance issues (like topics related to network, history, bookmark, speed, IO, memory, file system, build and document

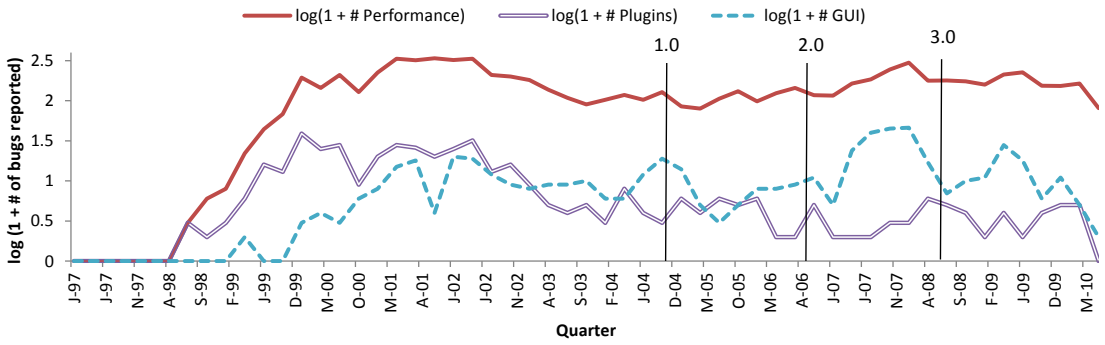

Figure 3.3: Comparison of the distribution of the number of bugs reported for all performance bugs and the two sub-categories of Plugin and GUI-related performance bugs.

structure) added up to the majority of performance bugs.

Figure 3.3 shows the distribution over time of plugin and user interface related bugs and their relation to all performance bugs. This figure is in log scale, so the peaks here denote large changes in the number of bugs reported. Between the release of version 2.0 and 3.0 of Firefox (shown in Figure 3.3), there was an increase in GUI-related performance bugs. We found that, version 2.0 included updates that were related to GUI such as a tabbed browsing environment, and an extensions manager, causing an increase to the number of GUI-related performance bugs.

For security bugs, as the number of security bugs was very small, the recovered LDA topics (such as "build-tinderbox related", "window-javascript related", and "user-cookie related") could not be grouped together into sub-categories. However, LDA results for security bug comments are included in Appendix A as a reference for any future research on security bug comments using topic models.

3.2 Results of our Case Study

Each subsection below discusses one of the three research questions that we studied using the Firefox data. For each question, we present the motivation behind the research question, the approach and a discussion of our findings. Table 3.2 summarizes the results of each of the three studied questions.

3.2.1 How Fast are Bugs Fixed?

Motivation: A project manager typically wants some types of bugs, like security and performance bugs to be assigned faster than others, because of their inherent importance in software quality assurance. In addition, he wants to pick the best developer for that bug, such that later on the bug does not need to be tossed to someone else to fix [40]. Finally, the bug needs to be fixed completely, such that it does not need to be reopened afterwards, prolonging the total time to fix the bug. Our study measures these characteristics for Firefox performance and security bugs.
Approach: Once a bug is reported to the bug repository, it browses through a complex life cycle (Figure 3.4), from the UNCONFIRMED state to the CLOSED state. Figure 3.4 shows the life cycle of a bug report in Bugzilla. In every state, it takes some time for the bug report to go to the next state.

We used four metrics for this question. First, we calculated the time to fix, i.e., the time between bug assignment (ASSIGNED) and the bug fix date (FIXED and then RESOLVED). This time was calculated for all bugs that went to the ASSIGNED state first and then from that, the bug went to the RESOLVED state directly (bolded arrow in Figure 3.4). Since some bugs are closed prematurely and have to be re-opened later (e.g., because of an incomplete fix), we measured both the total time to fix (total time

of each assignment to final fix) and the average time of each fix attempt. We used the t-test to compare the total and average times and the cumulative density function of the metrics for the three groups (performance, security and other bugs). As shown in sub-section 3.1.2, there is a large difference in the number of security, performance and other bugs. Hence, we used the Wilcoxon t-test in our study to ensure that the characteristics differences between bugs types are statistically significant [77].

The second metric that we used for comparison is the number of times a bug is reopened. In the ideal case, one would want a bug to be FIXED on the first try. In practice, a bug may be CLOSED, then REOPENED again. Reopened bugs take a considerably longer time to resolve, and hence increase the maintenance cost. For example, in the Eclipse platform 3.0 project reopened bugs take more than twice the time to resolve as a non-reopened bug [69]. In addition, an increased bug resolution time consumes valuable time from the already-busy developers, and reopened bugs may negatively impact the overall end-user's experience and trust in the quality of a software product.

The third metric that we used for our comparison is the bug triage time. Triage

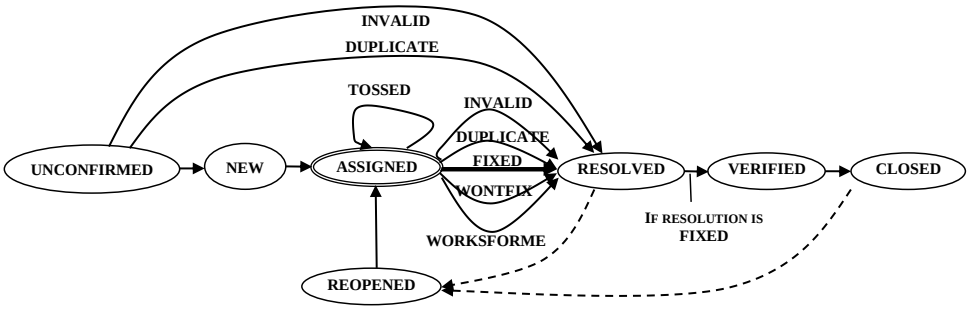

Figure 3.4: Life cycle of a bug [79].

time is considered separately from the time to fix a bug, as triaging and bug fixing are usually done by different people and comprise two different processes. Bug Triaging is the process of reviewing a reported bug in order to assign the right developer to fix it. Each project has a different strategy for this. Due to the volume of reports, reports submitted to the Mozilla bug repository are often triaged by quality assurance volunteers, rather than by the developers [6]. Ideally, one would want a bug to be triaged as fast as possible. We calculate the difference between bug reporting (UNCONFIRMED) and first assignment (ASSIGNED) as bug triage time. Although bug triage time affects the total bug fix time, it is also (and especially) important to select the appropriate person.

Our fourth metric measures the number of bug tosses, i.e., the number of times a bug was reassigned before being resolved completely [40]. Tossing may occur due to triaging inaccuracy, i.e., the developer assigned does not have the expertise to fix the bug. Another reason for tossing could be bug complexity, i.e., the assigned developer could only fix part of the bug and needed another developer to fix the rest of that complex bug.

Findings: Performance bugs take the longest time to fix. Figure 3.5 plots the "total time between every assignment and fix" (including the time between re-assignment and fix, when a bug is reopened). For example, a bug was assigned first to a developer and after t_1 minutes, the developer marked the bug to be fixed. If the bug was reopened and reassigned n times to a developer and the bug was fixed after $t_2, t_3, ..., t_n$ minutes, then we calculate the "total time between every assignment and fix" as $(t_1 + t_2 + ... + t_n)$ minutes. We use log-scale here to better show the differences between the three kinds of bugs. We can see a distinct difference for

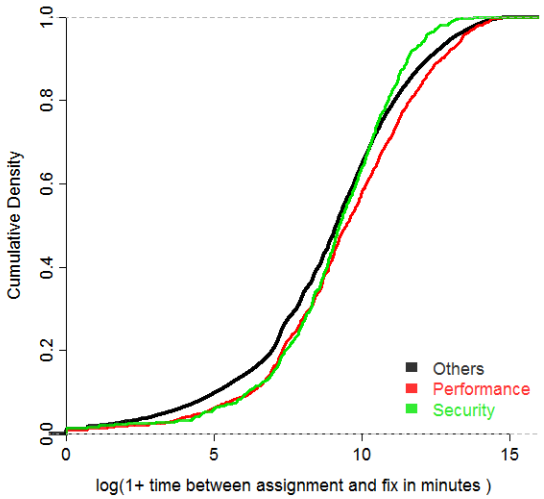

Figure 3.5: CDF (cumulative density function) of time between bug assignment and fix (total in log scale).

larger values. We ran t-tests to decide if the differences in mean between these three groups are statistically significant or not. We found a statistically significant ($p < 0.05$) difference between security and performance bugs. Security bugs generally take less time to fix than performance bugs. Other bugs take less time to fix than Performance bugs. However, the difference between security and other bugs is not statistically significant ($p = 0.0538$). According to the CDF plot, mean values and t-test result, we find that security bugs take the most amount of time to fix, followed by performance bugs and other bugs.

We also considered the average time to see how fast developers fix bugs. Considering the average time (each fix attempt's time), security bugs are fixed fastest, i.e., developers seem to respond quickly to them. On the other hand, performance bugs take the longest time for each developer. One possible explanation for this is that performance bugs often require changes across different subsystems of the architecture (see Q3). The difficulty of fixing such performance bugs might cause them to be

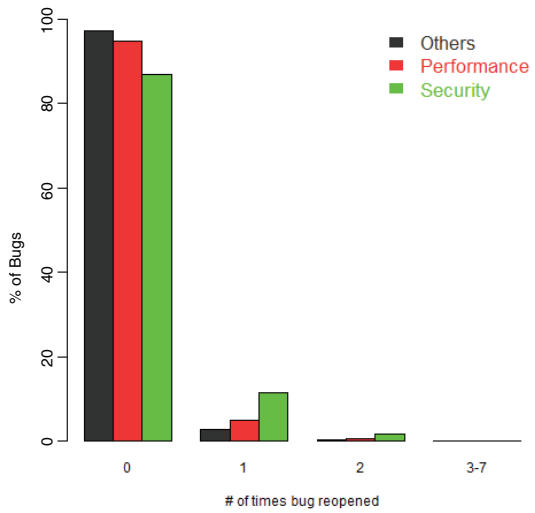

Figure 3.6: Histogram of number of times bug is reopened.

postponed to future releases or milestones.

Security bugs are reopened most frequently. Figure 3.6 shows the number of times a bug is reopened. As we can see, most of the bugs are never reopened (count_reopened = 0), but more than 10% of the security bugs are reopened at least once. Pairwise t-test comparison of the distribution of the number of times bugs are reopened shows that the difference between the three groups is statistically significant ($p < 0.05$). On average, security bugs tend to be reopened more than twice as often as performance bugs and performance bugs tend to be reopened almost twice as often as other bugs. There are various possible explanations for this. Security bugs might be more complex to solve and test compared to other bugs. Alternatively, while trying to fix security bugs as fast as possible (as seen above), there is a possibility that developers are not able to do sufficient testing for these bugs, which leads to these bugs being reopened later.

Security bugs are triaged faster than performance and other bugs. Statistical comparison of bug triage time reveals that performance bugs are not statistically different from other bugs in terms of bug triage time ($p = 0.15 > 0.05$), but security bugs have a smaller (70% smaller on average) triage time than others. A possible explanation for this might be that, to avoid the possibility of a security loophole being exploited, security bugs are kept secret (hidden) until the bug is fixed. Different software projects have different disclosure strategies [59].

Fast assignment is not identical to correct assignment. Our comparison of the number of bug tosses reveals that **security bugs are tossed more frequently than performance and other bugs.** As shown in Figure 3.7 and Table I, security bugs are tossed more than performance (166.7% more) and other bugs (371.2% more). These differences among the three groups are statistically significant. One common reason for tossing is incorrect assignment to a developer who does not own the defective source or may not have the expertise to fix the bug [40]. In any case, tossing events slow down the process and increase the total time to fix a bug.

While security bugs are re-opened and tossed more frequently, they are fixed and triaged the fastest.

3.2.2 Who Fixes Bugs?

Motivation: The task of finding the person with the most expertise to fix a bug is critical. In an empirical study on finding experts in a software development company, Ackerman and Halverson [4] observed that experience was the primary criterion engineers used to determine expertise [53]. For example, performance bugs are critical

Table 3.2: Mean and median metric values for the three research questions.

Metric			Mean			Median		
			Security	Perf.	Other	Security	Perf.	Other
Time	Time between assignment and fix (total time in minutes)		41,588.81	116,501.93	88,011.43	10,211	13,016	9,078
	# of times bug reopened		0.15	0.06	0.033	0	0	0
	Bug Triage time (in minutes)		57,347.80	208,983.55	193,700.40	4,098.03	8,904.97	6,714.02
	# of bug tossing		0.344	0.129	0.073	0	0	0
Person	# of developers assigned		1.203	0.503	0.343	1	0	0
	Experience	# of days	1,238.92	1,011.87	926.639	1,178	946	820
		# of prior bugs fixed	808.448	777.424	660.249	505	472	389
Bug Fix	# of lines changed		222.966	401.459	195.203	75.5	66	26
	# of files changed		3.236	8.396	4.527	1	2	2
	Complexity (entropy)		0.814	0.55	0.51	0.903	0.628	0.627

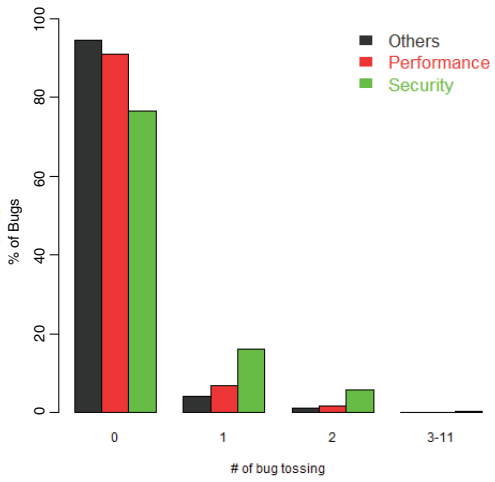

Figure 3.7: Histograms of the number of bug tossing.

in the sense that they require thorough knowledge of the software system, compilation tool chain, execution bottlenecks and memory layout. Similarly, fixing a security bug requires the understanding of possible security loopholes in the source code. In our case study, we are particularly interested in finding any relation between the performance or security bug fixers and the experience of the fixers.

Approach: To measure the number of developers whose expertise was needed to fix a bug, we count the number of unique developers assigned to a bug in its life time. For example, if a developer was assigned twice to the same bug (because of tossing or re-opening) we count this developer as one developer assignment.

We measure the experience of a developer fixing a particular bug using the following two metrics:

- Number of previously fixed bugs by the developer.
- Experience in days, i.e., the number of days from the first bug fix of the developer to the current bug's fix date.

For both metrics, if a bug was fixed by more than one developer (because of tossing and/or re-opening), we take the average experience of these developers.

A bug tracking system like Bugzilla uses an email address as developer identification, yet developers often has more than one email address [65]. For example, a bug could be fixed by John_Doe@mozilla.org and another bug could be fixed by John_Doe@gmail.com years ago, although both bug fixers are the same person. For all bugs, we use only the name part of the email address to deal with email aliasing, a technique commonly used for bug report and email mining [14, 15].

Findings: Security bugs require more developers to fix them than performance and other bugs. This follows from Figure 3.8 and a t-test on the data in Table 3.2 ($p < 0.05$). From the CDF plot in Figure 3.8, we see that 73% of other bugs and more than 62% of performance bugs are never assigned, compared to 14% of security bugs. Manual inspection of security bugs without developer assignment reveals that they are fixed very quickly, even before being assigned to someone. This

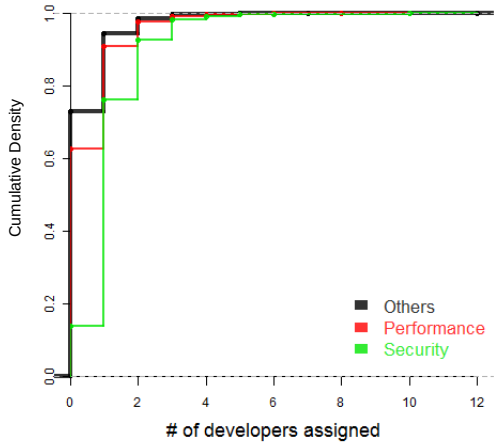

Figure 3.8: CDF of number of developers assigned

is probably due to the security bug disclosure strategy followed by Firefox [59]. Alternatively, some security bugs are fixed before they are entered in the bug repository by the developer. On the other hand, performance and other bugs are often not assigned because they are found to be a 'duplicate' of other bug reports that were fixed already, or found to be 'invalid' or 'resolved'. **Security or performance bug fixers are more experienced than fixers of other bugs.** From Figure 3.9 and Table I, we can say that, on average, security bug fixers and performance bug fixers have fixed a similar number of prior bugs, but fixed significantly more number of bugs than other bug fixers. In terms of experience days, on average, security bug fixers are 22.45% more experienced than performance bug fixers, and performance bug fixers are 9.19% more experienced than other bug fixers (for both cases, the t-test holds with p value < 0.05). In short, it appears that security and performance bugs require more experience to fix.

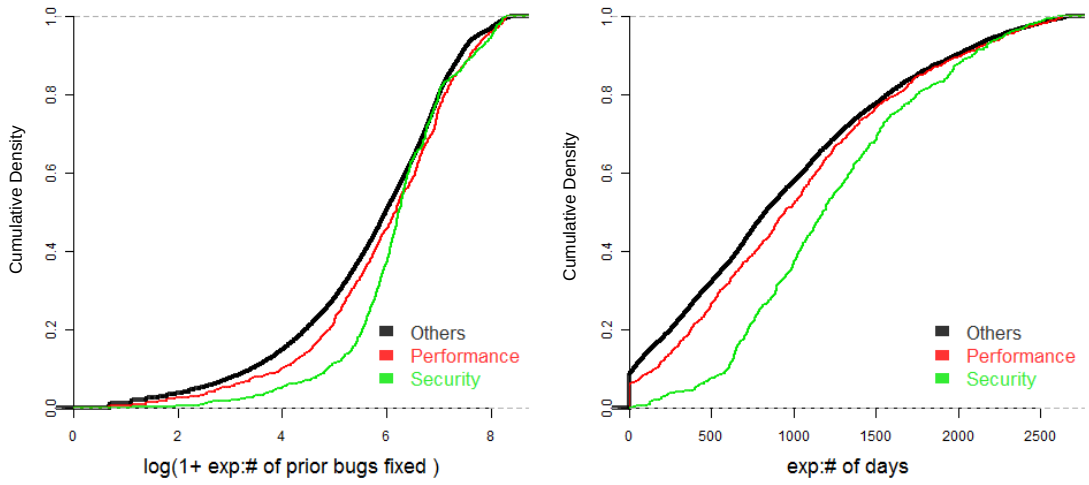

Figure 3.9: CDF of developer experience

On average, more developers are assigned to security bugs. Performance and security bugs require more experienced bug fixers than other bugs.

3.2.3 Characteristics of the Bug Fix

Motivation: So far, we have seen that security bugs require more bug fixers and are reopened more often than performance and other bugs. One possible reason for these findings might be the complexity of the bug fix. Consider an example of two bugs. In the fix of the first bug (the more complex bug), the developer had to change over a dozen files. When asked about the steps required to fix the bug, she or he may not recall half of them. For a bug fix for which the developer had to change only one file, recalling the changes required for that fix is much easier. In general, if we have a bug that required changes across all or most of the files of a software system, developers will have a hard time keeping track of all these changes.

Approach: To quantify complexity, we used three metrics:

– Total number of lines added/deleted.

– Total number of files changed for fixing a bug.

– Bug fix Entropy [36].

Bug tracking systems and bug reports do not contain the source code of a bug fix. To get the actual data related to a bug fix, one needs to link the bug report to the code repository (CVS). For this, we used the algorithm described by Sliwerski et al. [71]. We implemented their algorithm for the Firefox CVS code repository. As this approach relies on the developer revision comments having a bug id, we did not get the fix information for all of the bugs: 303 performance, 174 security and 7,800 other bugs could be linked to their source code changes out of 7,603 performance, 847 security and 286,759 other bugs.

Length of fix (number of lines added/deleted/edited) and spread across files (number of files changed) are straightforward to measure. To measure the bug fix entropy, we used the normalized Shannon Entropy, which is defined as: $H_n(P) = -\sum_{k=1}^{n}(p_k * \log_n p_k)$, where $p_k \geq 0, \forall_k \in 1, 2, ..., n$ and $\sum_{k=1}^{n} p_k = 1$. For a distribution P where all elements have the same probability of occurrence ($p_k = \frac{1}{n}, \forall_k \in 1, 2, ..., n$), we achieve maximum entropy. On the other hand, for a distribution P where only one element i has a probability of occurrence, we achieve minimal entropy (0).

For example, to fix a bug, three files A(5 lines), B(1 lines) and C(1 lines) are changed. We calculate the number of lines changed as the sum of the number of lines added and deleted. We define a file change probability distribution P as the probability that $file_i$ is changed for a bug's fix. For each file, we count the total

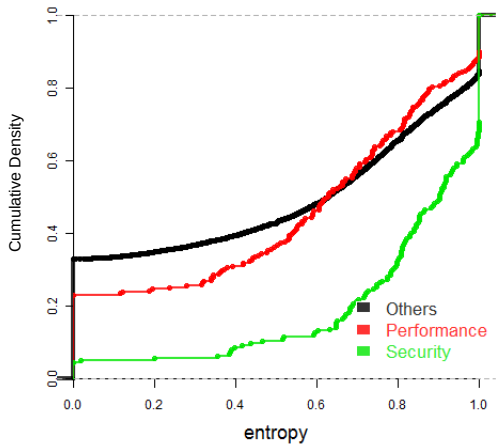

Figure 3.10: CDF of entropy

number of lines changed in that file and divide by the total number of lines changed for all files. Hence, in our example, we have $p(file_A) = \frac{5}{7}, p(file_B) = \frac{1}{7}, and\, p(file_C) = \frac{1}{7}$. Finally we can calculate the normalized Shannon Entropy for that bug's fix $H_n(P) = 0.725$. If another bug had the same total number of lines (7) changed in seven files, then Entropy would be, $H_n(P) = 1$. This means that the former bug fix is more concentrated, and hence less complex.

Findings: Performance bug fixes change more lines than fixes of other bugs. From Figure 3.11, Table 3.2 and the t-test, we find for fix size that the number of lines changed was significantly different for performance bugs in comparison to other bugs.

For the fix spread or number of files changed to fix a bug, we found that **fixing a performance bug requires change in more files than fixing a security bug.**

We found a more surprising result when we compared the entropy values. **Security bug fixes have the highest entropy**. There is a large difference between security bugs and the other two groups of bugs. On average, security bug fixes have

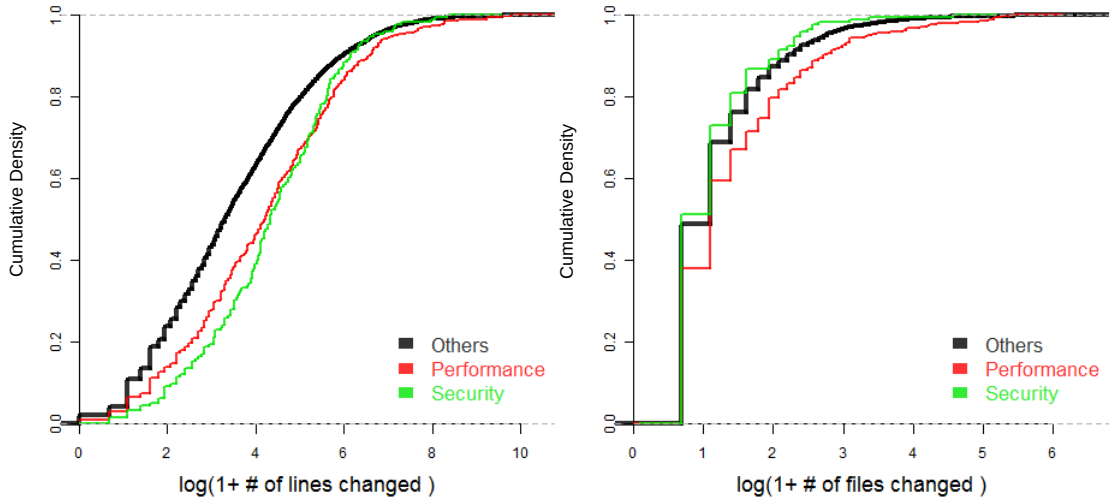

Figure 3.11: CDF of fix size

48% more entropy than performance bugs, and 59.6% more entropy than other bugs. This means that security bugs are much more complex to fix than performance and other bugs, since they require substantial fixes across multiple files. For example, we found that in the extreme case, for bug id 289940, there were changes across 296 files in the code repository. Further investigation showed that this bug revealed a security flaw that was extremely invasive: *"...The big part of this change is to mark all our internal events as trusted, where applicable. This will be done by changing the constructors of the various ns*Event classes .."*

Security bug fixes are more complex than performance and other bugs, yet they affect fewer files than performance bugs.

Table 3.3: Comparison of our qualitative study findings between Chrome and Firefox project

		Firefox		Chrome	
		Security	Perf.	Security	Perf.
Time	Fix time		+	+	
	# of reopening	+		+	
	# of tossing	+			+
People	# of developer assigned	+		=	=
	Experience	+			+
Fix	# of files changed		+	=	=
	Entropy	+		+	

'+' represents higher value for the type and '=' represents no statistically significant difference between performance and security bugs

3.3 Findings for Google Chrome and comparison with Mozilla Firefox

Similar to Mozilla Firefox project, we also qualitatively studied three types (performance, security and others) of bugs from Google Chrome web browser project. Details of the bug tracking system data used from Chrome project is provided in Subsection 4.2.1. A brief comparison of our findings is shown in Table 3.3. Only two metrics (number of reopening and bug fix entropy) showed consistent finding across Firefox and Chrome projects. Moreover, we did not find any metric for Firefox without a statistically significant difference between performance and security bugs. In Chrome project, we found two metrics ('number of developer assigned' and 'number of files changed to fix') that showed no statistically significant difference between performance and security bugs.

3.4 Threats To Validity

Our empirical study primarily focused on one system (the Firefox web browser). It is not clear whether our results generalize to other open source systems (possibly in different domains), or to commercial systems. We also studied Google Chrome bug repository data to verify if our study findings was consistent across projects from the same domain. We found that different bug types of Chrome project also have different characteristics. However, Firefox and Chrome project showed different findings. A brief description of the Google Chrome project results are discussed in Subsection 3.3.

We used heuristics on bug report titles and keywords to identify performance bugs. Since our study critically relies on bug classification, we statistically verified our heuristics for performance bug identification (section 3.1.1). A statistical sampling technique (with 95% confidence level and confidence interval of 10) showed that our heuristics have a high precision and recall.

Our data contains relatively few (847) security bugs in comparison to the large number of other (178,531) and performance (4,293) bugs. For the identification of security bugs, we are relying on the Mozilla Foundation Security Advisory and its links to Bugzilla. MFSA has been used in prior research as a source of security bug information [23, 70]. If there are any security bugs that were not linked to an MFSA advisory, they are treated as other bugs in our study.

Security bug reports may contain sensitive information explaining security loophole of the software that can be exploited. Therefore, some critical security bugs may be fixed in a different process that is not publicly accessible through the bug tracking system of the studied projects (Mozilla Firefox and Google Chrome). In our

quantitative study, we may have missed information about such critical bugs that may affect our study findings.

For collecting bug fix data using code repository revision comments, we used the information from revision comments using the algorithm described by Sliwerski et al. [71] (SZZ algorithm). Due to the limited number of revision comments with fix information and bug id, we could only get the associated fix code for 303 performance (20.2% of 1,503 performance bug fixes), 174 security (20.7% of 839 security bug fixes) and 7,800 other bug fixes (16.26% of 47,970 other bug fixes) for our study.

Finally, bugzilla fields for bug report time and bug history-related time do not necessarily correspond to the actual time. For example, if two or more bugs are assigned to a developer at the same time, and she fixes these bugs one by one, based on the bug history our analysis will find an incorrect 'time to fix' value for the bug that was fixed last. The "actual" time for fixing the bug starts once the developer is finished with the first bug and moves to the next one. Determining the actual time or effort in fixing a bug is an important step in project planning and management [51].

3.5 Chapter Summary

In this empirical study, we quantitatively analyzed the differences in time to fix, developer experience and bug fix characteristics between security, performance and the rest of the bugs to validate our conjecture that different types of bugs are different and hence quality assurance should take into account the bug type. We found that security bugs in Firefox behave differently than other bugs. Security bugs require more developers with more experience, but need less triage time and are fixed faster

than others. At the same time, security bug fixes are more complex than the fixes of performance and other bugs, and are reopened and tossed more than performance bugs.

To explain the results we have in our quantitative analysis described in this chapter, we perform a qualitative study on performance bugs, in the following chapter.

Chapter 4

A Qualitative Study on Performance Bugs

Chapter Overview: *Software performance is one of the important qualities that makes software stand out in a competitive market. However, in previous chapter, we found that performance bugs take more time to fix, need to be fixed by more experienced developers and require changes to more code than non-performance bugs. In order to be able to improve the resolution of performance bugs, a better understanding is needed of the current practice and shortcomings in bug life cycle. This chapter qualitatively studies a random yet qualitative sample of 400 performance and non-performance bug reports of Mozilla Firefox and Google Chrome across four dimensions (Impact, Context, Fix and Fix verification). We find that developers and users face problems in reproducing performance bugs and have to spend more time discussing performance bugs than other kinds of bugs. Sometimes performance regressions are tolerated as a trade-off to improve something else.*

Performance bugs are different from other types of bugs (i.e., security, usability, functionality) and require special care. As shown in the previous chapter, performance bugs take more time (at least 32% more on average) to fix, are fixed by more experienced developers and require changes to larger parts of the code than non-performance bugs [78]. Unfortunately, our analysis primarily considered high-level quantitative data like the overall bug fix time, the number of developers involved,

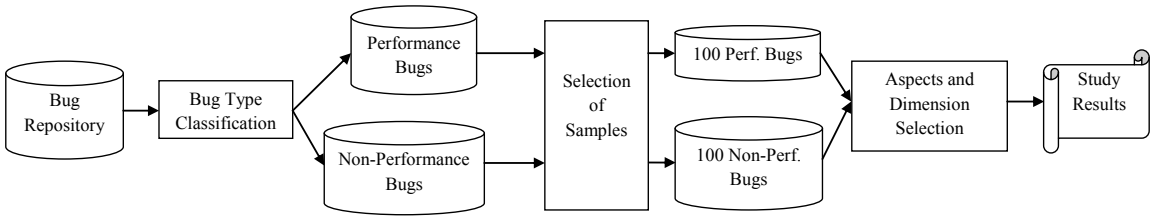

Figure 4.1: Overview of our approach to qualitatively study the performance bugs of a system.

and the size of the bug fix, but ignored qualitative process information like the content of discussions involved in the bug fix process. Although we were able to pinpoint differences between performance and other kinds of bugs, we could not fully explain these findings or make concrete suggestions to improve the fix process for performance bugs.

In order to understand the differences in the actual process of fixing performance and non-performance bugs, and potentially identify concrete ways to improve performance bug resolution, this chapter qualitatively studies performance bug reports, comments and attached patches of the Mozilla Firefox and Google Chrome web browsers. In particular, we studied a random yet representative sample of 100 performance and 100 non-performance bug reports and comments from each project (Mozilla Firefox and Google Chrome, 400 bugs in total) to discover the knowledge, communication and collaboration specific to fixing performance problems in a software system. We qualitatively compared the sampled performance and non-performance bugs across four dimensions and 19 sub-dimensions. We briefly summarize below our statistically significant findings for each dimension:

1. Impact on the stakeholder.

Performance bugs suffer from tracking problems. For example, Chrome has seven times more regression bugs that are performance-related than non performance-related. In Firefox, 19% of the performance bugs were fixed out of the blue without any concrete link to the patch or change that fixed them. Furthermore, in 8% of the performance bugs of Firefox, bug commenters became frustrated enough to threaten to switch to another browser, compared to only 1% for non-performance bugs.

2. ***Context of the bug.***

Despite the tracking problems and low reproducibility of performance bugs, 34% of performance bugs in Firefox and 36% in Chrome provided concrete measurements of e.g., CPU usage, disk I/O, memory usage and the time to perform an operation in their report or comments. In addition, twice (32%) as many performance bugs as non-performance bugs in Firefox provided at least one test case in their bug report or comment.

3. ***The bug fix.***

Fixing performance bugs turns out to be a more collaborative activity than for non-performance bugs. 47% of the performance bugs in Firefox and 41% in Chrome required bug comments to co-ordinate the bug analysis, while only 25% of the non-performance bugs required them. Some performance regression bugs remain unfixed on purpose.

4. ***Bug Fix Verification.***

We found no significant difference in how performance bug fixes are reviewed and verified compared to non-performance bugs.

Table 4.1: Taxonomy used to qualitatively study the bug reports of performance and non-performance bugs. Bold sub-dimensions show a statistically significant difference in at least one project.

Impact on the User	**Context of the bug**	**The Fix**	**Fix Verification**
Regression **Blocking** **WorksForMe** **after a long time** **People talking** **about switching**	**Measurement Used** **Has test cases** - by reporter ·with report - later **Contains stacktrace** - in report - in followup Has reproducible info - in report - in followup **Problem in** **reproducing** Duplicate Bug	**Problem discussion** **in comments** **Dependent on** **other bug** **Other bugs depend** **on this bug** Reporter provides some hint on the actual fix Patch uploaded by - reporter ·with report ·later - Others	Discussion about the patch Technical Non-Technical Review Super-review

Overview of the chapter: Section 4.2 explains our case study approach. Section 4.3 is divided into four sub-sections according to the dimensions considered in our study. Each sub-section of Section 4.3 discusses our findings for the different sub-dimensions found in the specific dimension. Section 4.4 summarizes the findings of this chapter with respect to the quantitative study on Firefox performance bugs in the previous chapter. In section 4.5, we discuss the threats to validity of our study. Finally, Section 4.6 presents the conclusion of the chapter.

4.1 Choice of Subject System

In this case study, we study the Mozilla Firefox and Google chrome web browsers, since these are the two most popular web browsers with publicly available data about their bug tracking systems. Similar to our quantitative study, we focus on the domain of web browsers, since for web browsers that support multiple platforms and system environments, performance is one of the major software quality requirements. For that reason, performance bugs are reported in both systems and are tagged explicitly

in the issue tracking systems of both projects. We considered the same Mozilla Bugzilla data that we used for our quantitative study (Chapter 3). For the Chrome web browser, our data was extracted from the Chrome issue tracker dump provided in the MSR challenge 2011 [68].

4.2 Study Design

This section presents the design of our qualitative study on performance bugs. Figure 4.1 shows an overview of our approach. For a particular project, we first extract the necessary data from its bug repository. Then, we identify the bug reports related to performance and random yet representative sample of 100 performance and 100 non-performance bugs. We then studied the sampled bugs to identify the different dimensions and their sub-dimensions that can be used to compare the bug fixing process of performance and non-performance bugs. Finally, we perform our analysis based on these sub-dimensions. This section elaborates on each of these steps.

4.2.1 Bug Type Classification

Using the same approach as we did for the Mozilla Firefox data in the quantitative study (Subsection 3.1.1), we found 510 performance bugs (1.13% out of 44,997 bugs) in the Google Chrome web browser project. In order to verify that the same heuristics that we used for finding performance bugs in Firefox work for this Chrome dataset, we again performed a statistical sampling with a 95% confidence level and a confidence interval of 10%. We selected and checked 81 random representative sample of performance (out of 510 bugs classified as being performance-related) and 96 random

Table 4.2: Confusion matrix for Chrome dataset

Actual →	Chrome	
Prediction ↓	**P**	**NP**
P	73	8
NP	2	94

representative sample of non-performance (out of 44,487 bugs classified as being non-performance) bugs [45]. Out of these 81 bugs classified as being performance-related, 73 were actual performance bugs, yielding a precision of 90.12 ± 10%. Out of the sampled 96 bugs classified as being non-performance, 94 were indeed non-performance bugs, yielding a recall of 97.33 ± 10% (see confusion matrix in Table 4.2). In other words, the heuristics seem to work well for Chrome as well .

4.2.2 Selection of Samples

For our 4 datasets of performance and non-performance bugs of Firefox and Chrome, the largest one is the set of Firefox non-performance bugs, with 176,057 bugs. With a 95% confidence level and 10% confidence interval, the sample size should be 96 bug reports. This means that if we find a sub-dimension to hold for n% of the bugs (out of these 96 non-performance Firefox bugs), we can say with 95% certainty that n ± 10% of the bugs contains that sub-dimension. The datasets with less bugs required less than 96 bug reports to be sampled to ensure this 10% confidence interval in the result. However, to simplify our discussion, we sampled 100 bug reports from each dataset, resulting in a confidence interval of 10% or less for each dataset.

4.2.3 Identification of bug report and comment dimensions

Before we can compare performance and non-performance bugs, we first need to establish the criteria to compare on. The different fields and characteristics recorded by bug repositories are a good start, but our study requires knowledge of how testers and developers collaborate on and think about (non-)performance bugs. This knowledge is part of the natural language content of bug comments and discussions, but we are not aware of any taxonomy or other work for analyzing qualitative report data. Hence, we first performed a manual study of the sampled data to identify such a taxonomy.

We studied each sampled bug report and tagged it with any word or phrase describing the intent or content of the bug comments and discussions. Each time new tags were added, older reports were revisited until a stable set of 19 tags was identified with which we could tag all reports. Afterwards, we analyzed the tags and grouped them into 4 clusters ("dimensions"): the impact of the bugs on stakeholders, the available context of the bug, the actual bug fix and the verification of the bug fix. The resulting dimensions and "sub-dimensions" (tags) are shown in Table 4.1 and are used to compare performance and non-performance bugs on. We discuss the sub-dimensions in more detail in our study results section (Section 4.3).

In order to determine, for a particular system, whether the percentage of performance bugs related to a sub-dimension (e.g., Blocking) is statistically significantly higher than the percentage of non-performance bugs related to that sub-dimension, we used the "joint confidence interval" or "comparative error". This measure is often used to compare two independent samples [29]. If the difference between the percentage of performance and non-performance bugs for a sub-dimension is greater than

their calculated comparative error, then the difference is considered to be statistically significant.

4.3 Study Results

Each subsection below discusses one of the four dimensions in our study on using Mozilla Firefox and Google Chrome data. For each dimension, we present the description of the dimension, the sub-dimensions and a discussion of our findings. All differences between performance and non-performance bugs mentioned in our findings are statistically significant unless stated otherwise. Any percentage value mentioned in our findings has a maximum confidence interval of 10%.

4.3.1 Impact on the stakeholder

Since different stakeholders are involved with a software system, e.g., developers, users, testers and managers, a bug can impact any of them. This impact can vary widely, from blocking a software release, to an annoying regression bug that pops up again, or to small annoyances in user experience. For example, release blocking bugs typically require immediate attention and priority from the developer assigned, while a small hiccup during start-up might be forgiven. In the worst case, failure to adequately fix a major bug can incite users to switch to a competitor's product when they are frustrated with a software problem. To understand the impact and severity of performance bugs and how stakeholders deal with that, we need to study this dimension.

Sub-dimensions:

Regression: A regression bug is caused by a certain event (like software upgrade,

another bug fix patch, or daylight saving time switch) that degrades a feature that used to be acceptable or re-introduces a previously fixed problem to the system [60]. This problem can be a performance problem or any other problem, like functionality or security-related. It also can be (re-)discovered via a failing test. Both bug tracking systems support the tagging of regression bugs, but do not enforce it. We identified the regression bugs from existing tagging and by studying the report details. The patch, build or version from which the regression started is typically also indicated.

Blocking: By "blocking release", we refer to bugs that prevent a release to be shipped, typically because they are showstopper bugs. These bugs are flagged in the 'Flags' field of a bug report in Firefox and labeled as 'ReleaseBlock' in Chrome. That label or flag also mentions the specific release version that is blocked. For example, in Chrome, the label mentions either 'Beta', 'Dev' or 'Stable' meaning that the bug blocks that release channel. If while working on a bug report, the project contributor/user feels that the bug has to be fixed before that release, she tags it as a release blocker.

WorksForMe after a long time: In our study, we found many bugs that were reported, confirmed to be valid, then closed after a long time with a WFM (Works For Me), Fixed or Won't Fix status. The 'WFM' (WorksForMe) status is specific to the Mozilla project and indicates that the problem mentioned by the reporter was not reproducible by the project contributor or others. Instead of 'WFM', the Chromium project uses 'Won't Fix' or 'Invalid' as a closing status. By "closed after a long time", we mean that a bug report is closed after several months and in many cases, more than a year.

People talking about switching: We counted the number of bugs where in the bug

comments, at least one user mentioned being unsatisfied with the software performance and thinking about switching to another browser because of that bug. Out of the millions of users of these browser, a comment from one of two people may seem to be not significant. However, since bug reporters are tech-savvy users, we can expect that each of them voices the opinion of many more other users in total.

Findings:

22% of the performance and 3% of the non-performance bugs are regression bugs [Chrome].

A similar phenomenon occurred in Firefox, but the difference was not significant (19% vs. 13%). Problems related to non-performance regressions include browser crashes, missing functionality, and unexpected behavior.

Since performance regressions are common problems, both projects have a dedicated framework to detect and report them. In Firefox, regression bugs are reported like any other performance bug, then identified and/or confirmed (and usually tagged) as a regression by the reporter and/or contributors. In Chrome, most of the regression bugs found in our study are manually reported by project members who detected the regression in the automated compile/test cycle. The report contains a link to automatically generated graphs that show the performance deviation between the build under test and the previous build. The reporter also provides some notes specifying the specific metric that showed the deviation as there are many metrics plotted in those graphs. Most of the Firefox regression bugs were reported after the software affected the user, while most of the Chrome regression bugs were reported even before the version was released.

Not all regressions are unexpected.

We found regression bugs that, although unexpected by the reporter, turned out to be known and tolerated by developers. Often, the performance bug was tolerated because of a trade-off with a more essential performance feature. In Chrome, we found the following comment in a regression bug (bug # 7992, comment # 3) that was closed with "WontFix" status.

> *"I talked to dglazkov and darin. This regression is expected and OK. dglazkov mentioned rebaselining the test, but I'm not sure that matters.*
> *The ... is a heavy user of the WebCore cache, and recent changes tilted the tuning toward memory usage instead of speed. If you look at the memory usage, you'll see a corresponding drop where the perf slows down."*

Performance improvement may cause regression.

We found different bug comments where a performance regression was introduced to the system while trying to optimize the performance. For example, in Firefox bug #449826, the bug was reported as a possible performance improvement suggestion, but the resulting performance ended up being much slower and still open since September, 2008.

14% of the performance bugs and 4% of the non-performance bugs are release blocking [Firefox].

A similar finding, but not significant, was found in Chrome (8% vs. 4%). Release blocking bugs have the highest priority to the developers.

19% of the performance and 6% of the non-performance bugs are marked as WFM/Fixed/Won'tFix after a long time [Firefox]

Although the final outcome of these three groups was different, in all of them bugs

have a traceability problem as we do not know how the bug was fixed, what caused the bugs, or why this bug will not be fixed. The reporter and other commenters who participated in the discussion of the bug report did not receive a response for a long time.

Possible explanation for this type of bugs can be, these bug reports might have been insufficient to work on a fix, or the bug might have been too complex. Ideally, some note on the state of the bug should have been recorded. This type of bug was also found in Chrome without any significant difference (10% perf. vs. 9% non-perf.) **8% of the performance and 1% of the non-performance bugs drove at least one user to threaten to switch to another browser [Firefox].**

In Chrome, 7% of performance and 2% of non-performance bugs are of this type and the difference was not statistically significant. These percentages do not include comments by users who are just frustrated with a bug. 7 users leaving Firefox might not seem much, but for each user that reports a bug, X other non-contributor users encounter the same issue and feel the same pain. For example, in a comment on Firefox bug #355386, (comment #10), the user said:

> *"... This bug is really bothering us and forcing us [his company] to recommend IE +SVG Adobe plugin to customers... "*

.

Surprisingly, this bug was only closed with WFM status after more than 2 years. In the Chromium project, a commenter said (bug #441, comment #6):

> *"I no longer use Chrome/Chromium for anything much and can no longer recommend to any of my friends or acquaintances or random people in the street that they use it. I still try it regularly in the hopes that the issue has been sorted out but as soon as the problem shows up I switch back to one of the others."*

Again, in this case, the problem was found to be not happening (WFM) anymore and closed with the status 'WontFix' after more than 10 months.

Users and developers perform performance comparisons with other browsers.

In both projects, we found that users and developers compare their product performance with competing products. While submitting a bug report in the Chromium project, the default bug report template suggests the reporter to provide information if the same problem was observed in other browsers. Mozilla bug reports usually also contain a comparison with other browsers. This is a common and expected practice irrespective of the bug type. It came as a surprise to us, however, to find bug reports in Google Chromium declared by the developers as "WontFix" only because the other popular browsers have the same performance issue or their browser at least was not slower than others. Users were not amused by this. For example, in Chrome, we found a comment (bug # 441, comment # 15) saying:

> *Thats a poor reason for closing a verifiable bug in something other than the rendering engine.*
> *Just becuase firefox consumes 100% cpu on something as basic as scrolling doesnt mean chrome should."*

One acceptable case of this pattern, however, happens when comparing Chrome functionality to that of the Safari browser. Since both share the same layout engine

"WebKit", many performance bugs had their root cause in webkit. In that case, the bug is identified as a "webkit" bug, reported to the webkit project's issue tracking system and the corresponding link is added to the Chrome bug report as a comment.

4.3.2 Context of the Bug

This second dimension captures the context available about a bug when reported in an issue tracking system. Bettenburg et al. did a survey on developers and users where they identified the factors that are considered to be helpful by the developers and reporters [13]. Both developers and reporters accepted that 'steps to reproduce' in bug reports are the most helpful information. Test cases and stack traces were next in the list. In the context of performance bugs, we wish to know if these kinds of information are used more/less/similarly in order to understand the performance bug reporting process. Access to more relevant context can reduce the performance bug fix time and thus the overall quality of software.

Sub-dimensions:

Measurement used: We counted the number of bugs that used any kind of measurements added in the report. By "measurement", we mean the use of any numerical value (e.g., CPU usage, disk I/O, memory usage) in bug report or comments to describe the problem.

Has test cases and contains stacktrace: Test cases are provided by the bug reporter or any other commenter to help reproduce the problem. We found different forms of test cases, i.e., specific web links, and attached or copy-pasted HTML/JavaScript files. In Chrome, very few bugs had test cases attached to them. Instead, they use problematic web links as test case. We found the use of attachments more convenient

as web links may become unavailable or are expected to be changed. Stack traces are also provided by the reporter and sometimes by other people in the bug comments. We counted the number of bugs where a stack trace was provided in the report. We also counted the number of bugs where the developer had to request a stack trace and the bug reporter or other commenter provided a stacktrace in a comment.

Has reproducible info and problem in reproducing: It is common practice in bug reporting is to provide a step by step process to reproduce a bug. This process should include any non-default preference, hardware requirement and software configuration needed to trigger the bug. However, we found many bugs where despite having the steps to reproduce the bug, people had substantial problems reproducing the bug and requested more information or suggestions for reproducing the bug. We counted the number of bugs where commenters mentioned such problems in reproducing.

Chromium bug reports contain information about whether the bug is reported by a project member or someone else. In Firefox bug reports, we could not see any such information. Hence, we counted the number of performance and non-performance bugs reported by a project member only for Chrome.

Duplicate bug: Bug reports contain information about duplicate bugs, i.e., bugs that have the same problem as any other reported bug but the reporter did not notice that or could not recover that in the issue tracking system before reporting. We found that duplicate bugs are identified and marked by the project contributors and sometimes by the reporter. Firefox has a field "duplicates" where all the duplicate bugs are listed. In Chrome, we identified duplicate bugs from the bug status "Duplicate" or from comments like, *"Issue ... has been merged into this issue"*. We counted the number of bugs with duplicates for all sampled performance and non-performance

bugs in our study.

Improvement suggestion: We counted the number of bugs that were about improvement (both performance and non-performance) of the software, not about defects. We identified bugs related to improvement by checking if the report is about speed, resource utilization, usability or code quality improvement instead of a specific bug.

Findings:

34% of the Firefox and 36% of the Chrome performance bugs contain performance measurements.

More than one third of the performance bug reports and/or comments provided measurements of different performance metrics. We observed an interesting difference in the use of measurements between Chromium and Firefox. In Firefox, the measurements included in bug reports were numerical, i.e., CPU usage, disk I/O, memory usage, time to perform an operation, or profiler output. In addition to these numerical values, Chrome performance bug reports also contained the metric values calculated using Chromium's performance test scripts that run on "buildbot", the continuous integration server used by Chromium. Although Mozilla also had test suites for the "tinderbox" [65] continuous integration server, we could not find any bug report in our sample that referred to this test suite.

As expected, we could not find any non-performance bug that had any kind of measurement mentioned in its bug report, since it is hard to quantify a functional bug.

32% of the performance and 16% of the non-performance bug reports or comments contain a test case [Firefox].

Most of the time (more than 70% of the time for both bug types and projects), test cases are uploaded by the reporter. In Chrome, the difference was not statistically significant (21% performance vs. 16% non-performance).

10% of the performance and 1% of the non-performance bugs contain a stack trace in the report or a comment [Chrome].

In Chrome, we could not find a statistically significant difference between performance (13%) and non-performance bugs (8%).

15% of the performance and 6% of the non-performance bugs had comments related to problems with reproducing the bug [Firefox].

Insufficient information provided by the reporter or the unavailability of a test case is one of the most common reasons behind being unable to reproduce any bug. Moreover, for many performance bugs, the problem was caused by the use of a specific (version of an) extension or plugin of which the reporter could not collect sufficient information. We often found that in the bug comments, commenters guide the reporter by giving instructions and web links on how to collect the required context that will help the commenter and other people to reproduce the bug. We also found intermittent problems hard to reproduce. In Chrome, 16% of the performance and 8% of the non-performance bugs had comments related to problems in reproducing, but this difference between performance and non-performance bugs was not statistically significant.

58% of the performance and 35% of the non-performance bugs are reported by a project member [Chrome].

One possible reason of this difference may be that performance regression bugs found by an automated test script are always reported by a project member. We

found that out of the 22% performance regression bugs in Chrome more than 86% were reported by a project member, which shows a larger involvement of project members in reporting performance bugs.

In both projects, many performance bugs are caused by external programs

We found that 11% of the performance bugs in Firefox and 14% of the performance bugs in Chrome are caused by a program that was not developed within the project. Investigations of these bugs showed performance problems caused by an "add-on" or "plugin" (e.g., flash), a single web application/page (e.g., yahoo mail or gmail) or any other third party software in the system (e.g., antivirus).

In Firefox, comment # 1 of bug # 463051 in November, 2008 mentioned,

> *"Report problems caused by external programs to their developers. Mozilla can't fix their bugs. ..."*

The bug was marked as resolved with resolution "invalid". In the next comment, which was submitted more than two years later, we found a comment saying,

> *"We're now tracking such bugs. This doesn't mean it's something we can fix, merely something we hope to be able to point vendors to so they can investigate. This is an automated message."*

After that comment in March, 2011, the bug was assigned a new QA (Quality Assurance) contact and since then, the status of that bug is "unconfirmed".

The Chromium issue repository kept track of these issues from the beginning, to support the QA team in collaborating with the users facing the reported problem.

Non-performance bugs have more duplicate bugs reported than performance bugs [Firefox].

In Firefox, the difference was significant (26% performance vs. 40% non-performance), while we could not find a statistically significant difference in Chrome (24% performance vs. 32% non-performance).

Although steps to reproduce are used in most of the bug reports, we could not find any statistically significant difference between its use in performance and non-performance bugs. We also could not find any significant difference for the sub-dimension "improvement suggestion".

4.3.3 The Bug Fix

Process metrics like the response time of the fix process can be used to measure software quality [46]. However, in order to improve the performance bug fixing process and overall quality, we need to know how, given the context of the second dimension, a performance bug is fixed and what (if any) makes the performance bug fixing process different from other bugs in practice.

Sub-dimensions:

Problem discussion in comments: We studied the bug comments where people discussed the problems related to the bug report and the uploaded patch for that bug. To distinguish between problem discussion and patch discussion in Firefox, we considered all comments before the first patch upload as problem discussion, and all comments after the first patch upload as patch discussion (see Bug Fix Verification dimension). As Chrome uses a different tool for patch review than the issue tracking system, we considered all bug comments as problem discussion.

Dependent on other bug or other bug depends on the bug: We also studied the dependency information provided in the issue tracking system. When a bug's fix depends on the resolution of another bug, this dependency is provided in both issue tracking systems in two different fields ("Depends on" and "Blocking").

Reporter provides some hint on the actual fix: We found many bugs (specially improvement suggestion bugs) where the reporter had an idea about the fix and provided the hint and sometimes the patch with the bug report. We counted the number of performance and non-performance bugs of this type.

Patch uploaded by: We found bug patches that were uploaded by its reporter, sometimes within a few minutes after reporting. We counted the number of bugs where the patch was uploaded by the reporter (immediately after reporting/later) and by others.

Findings:

Performance bugs are discussed more than non-performance bugs in order to develop a fix.

47% of the performance bugs of Firefox and 41% of the performance bugs in Chrome required a discussed in the bug comments, while only 25% of the non-performance bugs in both projects had such a discussion. By discussion, we mean the process of exchanging ideas in order to understand a problem, identify the possible fix, and reach a decision. We did not count those bugs that had only comments saying something like, "it happens to me too" rather than trying to find out the reason for that problem. As such, this finding suggests that performance bugs are harder to resolve, or at least require collaboration between multiple people.

Table 4.3: Number of Blocking and Dependent bugs (bold numbers have a statistically significant difference).

Project	Type	Blocking	Depends on
Firefox	**Perf.**	**44**	**42**
	Non-perf.	**27**	**15**
Chrome	Perf.	2	6
	Non-perf.	6	4

Performance bugs are more dependent on other bugs and more performance bugs are blocking than non-performance bugs [Firefox].

Table 4.3 shows the dependency information found in our studied bugs. When 'B' blocks the resolution of bug 'A', bug 'B' is marked as blocking bug 'A' and bug 'A' is marked as depending on 'B'. Only Firefox is showing a difference between performance and non-performance bugs (42% vs. 15% for depends on and 44% vs. 27% for blocking) with statistical significance. Very few dependency information was found in Chromium project and the difference was not statistically significant.

Patches are not always used for fixing a bug.

We found that many of the performance bugs in Chromium project are reported by a project member who observed a performance deviation in the automated performance test result. The performance test scripts running on "buildbot" compare the performance metric against a performance expectation baseline. The performance expectations are written in a file ("/trunk/src/tools/perf_expectations/perf_expectations.json") that has the expected 'improve' and 'regress' values. By comparing these values to the actual value, the performance regression testing script identifies both "performance regression" and "unexpected speedup" (bug # 18597, comment # 11). We found many patches related to performance bugs where this expectation baseline file was

patched to readjust after a definitive performance speedup or an accepted regression. For accepted regressions, the corresponding bug is no longer a bug, but expected behavior. Moreover, some patches are only intended for troubleshooting. For example, in bug # 34926 comment # 9, a patch was uploaded to determine if another revision caused the performance deviation.

For the sub-dimensions "reporter has some hint about the fix" and "patch uploaded by", we could not find any significant difference between performance and non-performance bugs.

4.3.4 Bug Fix Verification

After a software fix has been proposed, it needs to be verified by reviewers to ensure software quality. Basically, reviewers comment on things like the approach, algorithm used, and code quality in order to ensure that the patch really fixes the bug and the fix does not introduce any new bug. Sometimes, depending on the importance of the bug and complexity of the fix, a bug fix may require a more rigorous verification effort.

Sub-dimensions:

Discussion about the patch: After a patch is uploaded and linked to the issue tracking system, people can discuss about the patch in order to verify the fix by the patch. In the Firefox Bugzilla, patch discussion appears in the bug comments (the same page as the bug report). In Chrome, we found this discussion on the Chromium code review page for that patch (a separate page whose link is posted as a bug comment). Some discussions focus solely on the code changes in the patch. We call these technical discussions. Other discussions are not source code specific, i.e.,

Table 4.4: Comparison of findings about performance bugs between our previous qualitative study (Chapter 3) and this study for Mozilla Firefox project.

Quantitative Study	Qualitative study
Require more time to Fix	1. WorksForMe (WFM) after a long time, 2. Problem in reproducing 3. More dependencies between bugs 4. Collaborative root cause analysis process
Fixed by more experienced developers	1. More release blocking 2. People switch to other systems
More lines of code changed	N/A

they discuss about whether the problem is still observed by others, and what changes people can see after the patch is used. We call these non-technical discussions.

Review and super-review: In both projects, the code was reviewed by one or more code reviewers. Moreover, we found the use of the term "super-review" in Mozilla. A super-review is performed by a super-reviewer (a list of reviewers indexed by area) and is required for certain types of critical changes mentioned in Mozilla's super-review policy [3].

Findings:

For none of the projects and sub-dimensions, performance bug reports behave statistically significantly different than non-performance bugs.

4.4 Discussion

In the previous chapter, for the Mozilla Firefox project, we found that performance bugs take more time to fix, are fixed by more experienced developers, and that more lines of code are changed to fix performance bugs. Based on the qualitative findings in

this chapter (summarized in Table 4.4, we are now able to, at the minimum, provide more context about our previous findings and, likely, provide a strong indication about the actual rationale behind those quantitative findings.

Related to the finding that Firefox performance bugs take more time to fix, we found that for Firefox, people face problems in reproducing performance bugs and have to discuss them to find the root cause of these bugs. These collaborative activities may require more time in a project of global scale like Firefox. Moreover, we also found that performance bugs have more dependencies on other bugs, which implies that performance bug fixing may be delayed because of the time to fix the dependent bug(s) first. On the other hand, we also found indications that our measurements of the time to fix a bug might have been inflated. Indeed, 19% of the performance bugs in Firefox are fixed after a long time (in the order of months, sometimes years) without any trace of related fixing process. They were basically fixed as a side-effect of some other bug. Since the corresponding bug report has been open (and probably forgotten) for so long the unnatural fix time of these bugs may cause the average fixing time to be higher than it should be.

Related to the finding that performance bugs are fixed by more experienced developers in Firefox, we found in this chapter that a higher percentage of performance bugs is release blocking than non-performance bugs in Firefox. Release blocking bugs are known to be of the highest priority and this may prompt projects to assign experienced developers to fix performance bugs. Furthermore, the difficulty to reproduce performance bugs and fix them also suggests that experts are better equipped to fix performance bugs.

Finally, we did not find direct qualitative support for our finding that performance

bug fixes are larger than non-performance fixes. However, our findings that performance bugs have many dependencies and require experienced developers to fix them could give some indications that performance bugs need more system-wide knowledge instead of knowledge about one component. More analysis is needed to further explore this possibility.

4.5 Threats To Validity

Since this is a qualitative study, there may be some human factors and subjectivity in our bug analysis, since it is very difficult to prevent or detect researcher-induced bias in such a qualitative study. However, we took care to consider all available bug data sources and to double-check findings when in doubt.

There may be more dimensions and/or more sub-dimensions in the selected dimensions other than the ones that we selected, since our selection is based on the sampled data in Firefox and Chrome. We iteratively identified tags until a stable set of tags was found that could cover all sub-dimensions of sampled reports. Conceptually, the (sub-)domains cover most of the qualitative aspects of bug reports, and preliminary analysis on Firefox and Chrome projects support the claim.

Our qualitative study focused on two projects. It is not clear whether our findings generalize to other open source projects or to commercial projects. However, we have selected the two most popular open source projects (Mozilla Firefox and Chrome) from the browser domain, since performance bugs matter in this domain and we had access to all relevant data.

We used heuristics on bug report titles and keywords to identify performance bugs. Since our study critically relies on bug classification, we statistically verified

our heuristics for performance bug identification. A statistical sampling technique (with 95% confidence level and confidence interval of 10) showed that our heuristics have a high precision and recall.

4.6 Chapter Summary

Mozilla Firefox and Google Chrome are the two most popular open source web browser projects with millions of users and hundreds of developers and contributors around the world. We studied sampled performance bugs in these two projects to observe how project members collaborate to detect and fix performance bugs.

Performance bugs have different characteristics, in particular regarding the impact on users and developers, the context provided about them and the bug fix process. Our findings suggest that in order to improve the process of identifying, tracking and fixing performance bugs in these projects:

- Techniques should be developed to improve the quality of the "steps to reproduce" in the performance bug reports.
- More optimized means to identify the root cause of performance bugs should be developed.
- Collaborative root cause analysis process should be better supported.
- The impact of changes on performance should be analyzed, e.g., by linking automated performance test results to commits, such that performance bugs no longer magically (dis)appear.

One of the findings in this chapter is that, despite efforts for improving software performance by the developers and testers, customers get frustrated and threaten to

switch to competing software. Motivated by this finding, we wanted to study this in more detail, i.e., is it possible that some users are put at a disadvantage which goes unnoticed by the testers and developers?

Chapter 5

A Large Scale Empirical Study on User-Centric Performance Analysis

Chapter Overview: *Measuring the software performance under load is an important task in both test and production of a software development. In large scale systems, a large amount of metrics and usage logs are analyzed to measure the performance of the software. Most of these metrics are analyzed by aggregating across all users to get general results for the scenario, i.e., how individual users have perceived the performance is typically not considered in software performance research and practice. To analyze a software's performance, user's perception of software performance metrics should be considered along with the scenario-centric perspective of system tester or operator.*

In Chapter 4, we found that some users in both the Firefox and Chrome projects who were unsatisfied with the software performance despite continuous effort by the project members to detect and fix performance issues, and improve the software performance. Performance problems faced by some users may have passed unnoticed by the project members in their performance tests and analysis. Motivated by this, we wanted to find out, if a user-centric performance analysis could identify these unsatisfied users to the project members that would not be noticed when using an overall performance analysis.

Measuring the software performance perceived by the user during test or production of a software allows an organization to adjust the deployment of a software system in order to meet the performance expectations of its users [38, 55]. Since performance measurements gather gigabytes of log and performance counter data, performance analysts typically aggregate this data across all users, or clusters of users with the same quality of service (QoS), to get more manageable information. For example, a system can be analyzed to see if 99% of the response time for a test scenario was under 2 milliseconds.

Aggregation of performance data loses knowledge about how individual users are affected by the performance. A large software system may have a low average system response time, which suggests that all users are satisfied with the performance, while from the user perspective, it may be possible that some thousands (out of millions) of users are always unsatisfied because of persistent high response time (i.e., because of software compatibility issues at the user's end). In a competitive market with other options for the user, user perceived quality is important for user satisfaction and user loyalty to the service provider [20]. Moreover, if these unsatisfied users are the major users, it can be disastrous for a business.

User perceived quality has been studied in the field of software system design, usability analysis, product and process quality [18, 24, 32, 54, 72]. The user's perspective of quality can be different from the developer or manager's aggregated perspective. Performance can be viewed as one of the characteristics of software quality [10].

In this study, by the term "user-centric view", we refer to the view of software performance data aggregated for individual users. We compared individual user performance data (user-centric) with the aggregated performance data (scenario-centric)

to see whether or not the user-centric performance analysis data provides the same information as the scenario-centric data. We could not use the same Mozilla Firefox and Google Chrome projects in this study as data from the performance test that runs on these projects was not available. In this controlled case study of two large enterprise systems, and one small open-source e-commerce system's performance test data (with simulated load), we calculate scenario-centric and user-centric metrics to address the following three research questions:

Q1*. *How does the user-centric performance experience differ from the scenario-centric one?

One of the enterprise systems showed major performance improvements for 2 (out of 4 analyzed) use case scenarios while the scenario-centric comparison could not identify these improvements. In the open source e-commerce system, user-centric analysis in 3 (out of 3) scenarios showed that 40% of the users in each scenario had a bad performance in at least 30% of their requested operations, while the scenario-centric analysis showed that on average, 13.11% to 20.80% of these operations behaved badly in these three scenarios.

Q2*. *How does the user and scenario-centric performance evolve over time?

When a system is running for a long time, it may encounter performance problems that could not be visible if it was running for a short amount of time. 2 (out of 4) and 5 (out of 6) scenarios in the two enterprise software systems and 1 (out of 3) scenarios in the open source system showed a different time trend between the scenario and user perspective. These differences suggest that

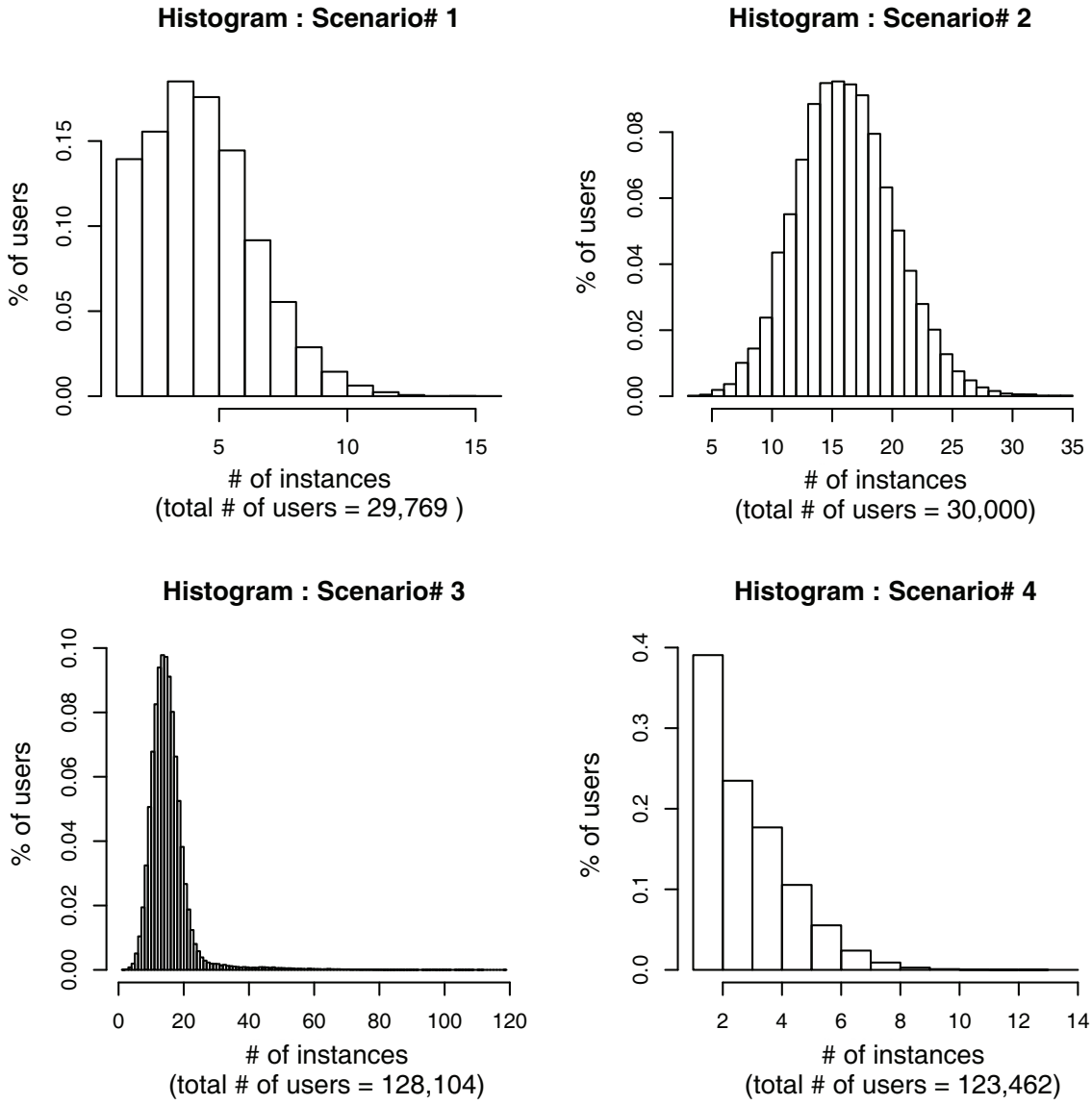

Figure 5.1: Enterprise System 1 - Histogram of number of scenario instances per user

scenario-centric performance trends do not necessarily reflect the performance trends perceived by user.

Q3. How consistent are the user and scenario-centric performance characteristics?

A system may sometimes perform outstandingly good and sometimes very poorly while the users in the system may expect a moderate and consistent performance from the system all of the time. In user-centric analysis, we found scenarios where a large percentage of users who had an inconsistent performance while the system's performance looked consistent in scenario-centric analysis (on average).

Overview of the chapter: The chapter is organized as follows. Section 5.1, discusses the study design and data source for our study. Section 5.2 explains the motivation, approach and results for our three research questions. Section 5.3 discusses the threats to validity, and Section 5.4 presents the conclusion of the chapter.

5.1 Study Design

In this section, we discuss our three case studies in which we performed both user-centric and scenario-centric performance analysis. User-centric analysis measures the metrics from the individual user's perspective and scenario-centric analysis measures the metrics for a scenario across all users. Table 5.1 shows an overview of the three projects in our study, i.e., two commercial enterprise systems (ES) and one open source e-commerce system, the Dell DVD store (DS2). We studied the performance load test data for these three systems. In performance load test, load generator

software(s) simulate the user behavior by sending simulated requests to the software system under test for different use-case scenarios. For each use-case scenarios of the software system, repeated requests are sent and monitored to ensure that the software do not have any performance issue, i.e., insufficient cache size and memory leak. For each system, we considered the most popular use cases. For each instance of such a use case, we collected the user id and the response time (i.e., the time between start and end). All the metrics used in our analysis are derived from this response time and user id data.

Table 5.1: Properties of the three case studies.

Factor	**Enterprise System 1**	**Enterprise System 2**	**Dell DVD store**
Functionality	Telecommunications		E-commerce
Vendor's Business Model	Commercial		Open-source
Size	Ultra Large	Large	Small
Complexity	Complex	Complex	Simple

Enterprise System 1 (ES 1): The first system for our case study is an ultra large scale distributed system for which we had access to the performance regression test log data. A test was run on two versions of the system with the same test operational profile (i.e., a realistic distribution of user input requests). Any performance deviation found in the new version's performance is reported by the tester to the developer. One hour of test log data was collected. Log lines contain log messages that can be grouped into sequences per user session and mapped to a use case using log sequence patterns known to the system experts [9, 21, 43].

We compare the old and the new version of the ES 1 to verify if the scenario-centric performance analysis differs from the user-centric performance analysis. Out of around 100 scenarios, we considered the four most occurring ones as the four major use-cases of this system. Other scenarios had a too low frequency of occurrence to comment on their performance. Every scenario's first and last log line's time-stamp was used to calculate the response time for that scenario. Ideally, every user in these tests is expected to perceive a similar response time.

Enterprise System 2 (ES 2): The second system in our case study is a large scale enterprise system of which we could use the test log data. The test was run on one version under development and 10 hours of log data was collected and analyzed. We used a similar log sequence analysis approach as we did for ES 1. Out of 1,092 scenarios, we considered the top 6 most popular log sequences.

Dell DVD store (DS2): The third case study was performed on the Dell DVD store (DS2). DS2 is an open-source three tier simulation of an e-commerce website [28]. We set up the DS2 test server in a lab environment using two Pentium 3 servers running Windows 2008 and Windows XP with 512MB of RAM. The first machine is the Apache Tomcat web application server [73] and the second machine is the MYSQL 5.5 database server [57]. The load for this test was generated using the DS2 web load driver running on a Pentium 4 machine with 2GB of RAM. The load generator simulates multiple users by sending HTTP requests to the web server for each user, the web server processes the request by communicating to the database server, then replies to the user (load generator) again.

We ran the test for more than 11 hours and monitored the server resource utilization logs to identify the initialization period of the system. We removed the initial

1 hour log data to remove the system initialization performance from our study and concentrated on the normal operation of the system. Details of the Dell DVD store (DS2) load generator configuration and other graphs are provided in Appendix B to help any future replication study. We instrumented the DS2 driver program that generates the load to the web application to create log files containing one line for each user's HTTP request+reply (i.e., for each instance of every use case scenario). As a simple e-commerce system, DS2 has four scenarios: Create new user, Login to the system, Browse products and Purchase. As the scenario 'Create new user' occurred only once for each user, we dropped it from our study.

In total, our study contains 4 use-case scenarios from ES 1 (2 versions), 6 use-case scenarios from ES 2 and 3 use-case scenarios from the Dell DVD store e-commerce system.

5.2 Results of our Case Study

For each question, we present the motivation behind the research question, the approach and a discussion comparing the results of the user-centric approach with those of the scenario-centric approach for analysis.

5.2.1 How does the user-centric performance experience differ from the scenario-centric one?

Motivation: A user's initial experience to the system (i.e., the experience of the first few operations performed by a user) is very important to the user's confidence in the system's performance and to keep him using the system. In a competitive market, bad experience of usage may cause the user to ask for a refund, look for another new

Table 5.2: Enterprise system 1 - scenario-centric Comparison of performance

Scenario #		Old	New
1	Mean	227.34	226.1
	Median	204	203
	% of bad instances	12.65	11.32
2	Mean	410.62	425.89
	Median	406	406
	% of bad instances	16.48	0.21
3	Mean	44.82	42.25
	Median	46	32
	% of bad instances	7.19	0.37
4	Mean	8.24	7.14
	Median	0	0
	% of bad instances	5.03	4.64

service provider, repeatedly call to system support lines or ask for technical support.

A user's overall experience (i.e., the experience across all operations by a user during our test analysis time frame) explains whether the initial experience was representative for the full experience. For example, if there are some users who have a persistent problem in their network or system configuration, they will always perceive a bad experience. In case of a relatively small number of such unlucky users, the scenario-centric analysis of the system's performance may miss such information, although the performance problem of those users can be important. A good run from the scenario-centric perspective may actually be a bad run from the user-centric perspective.

Approach: Initial Experience: For the initial experience, we considered the

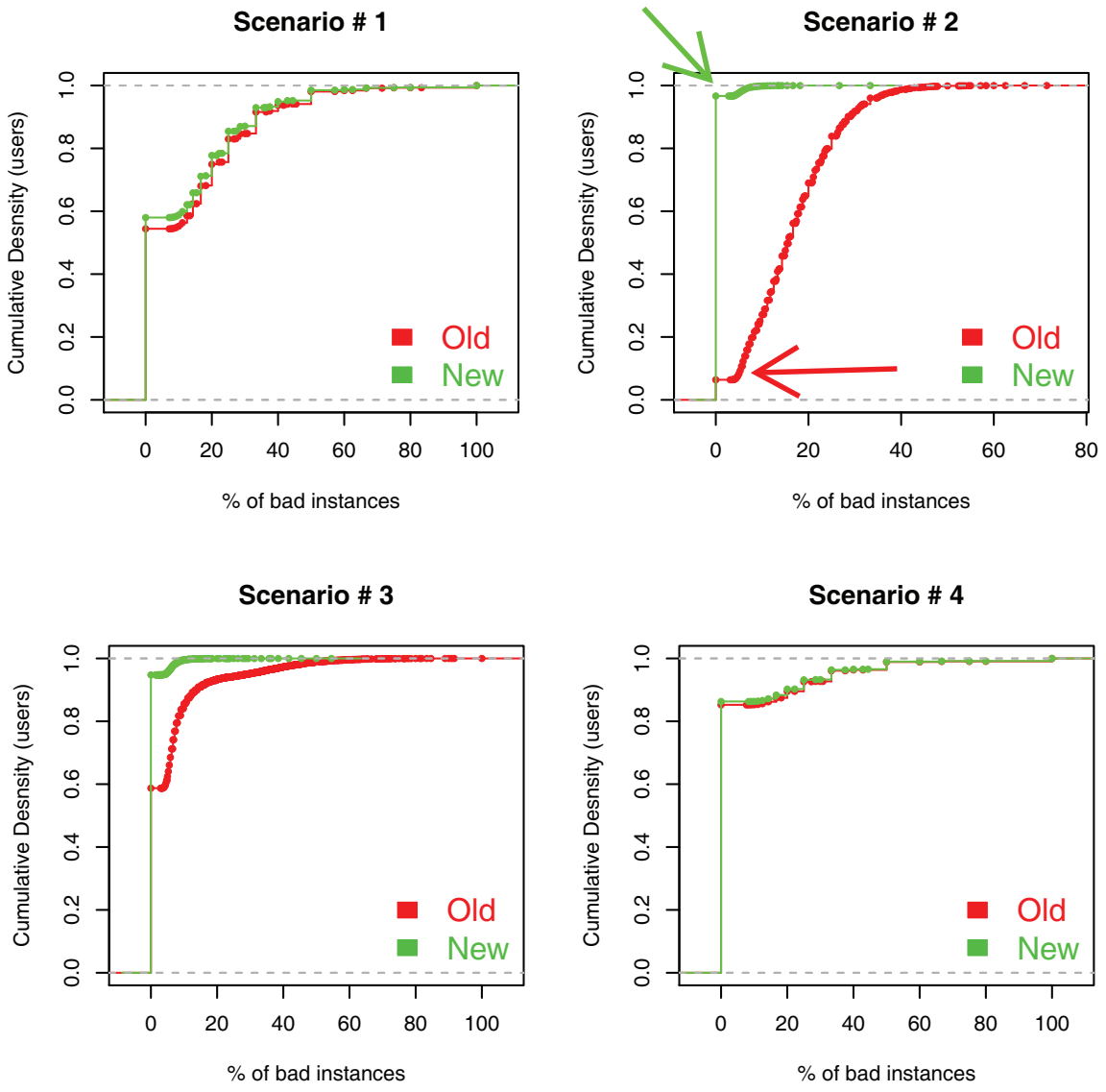

Figure 5.2: Enterprise System 1 - Cumulative Density Plot of user Overall Experience in % of bad instance

arrows in Scenario # 2 are indicating the % of users who had 0% bad instances

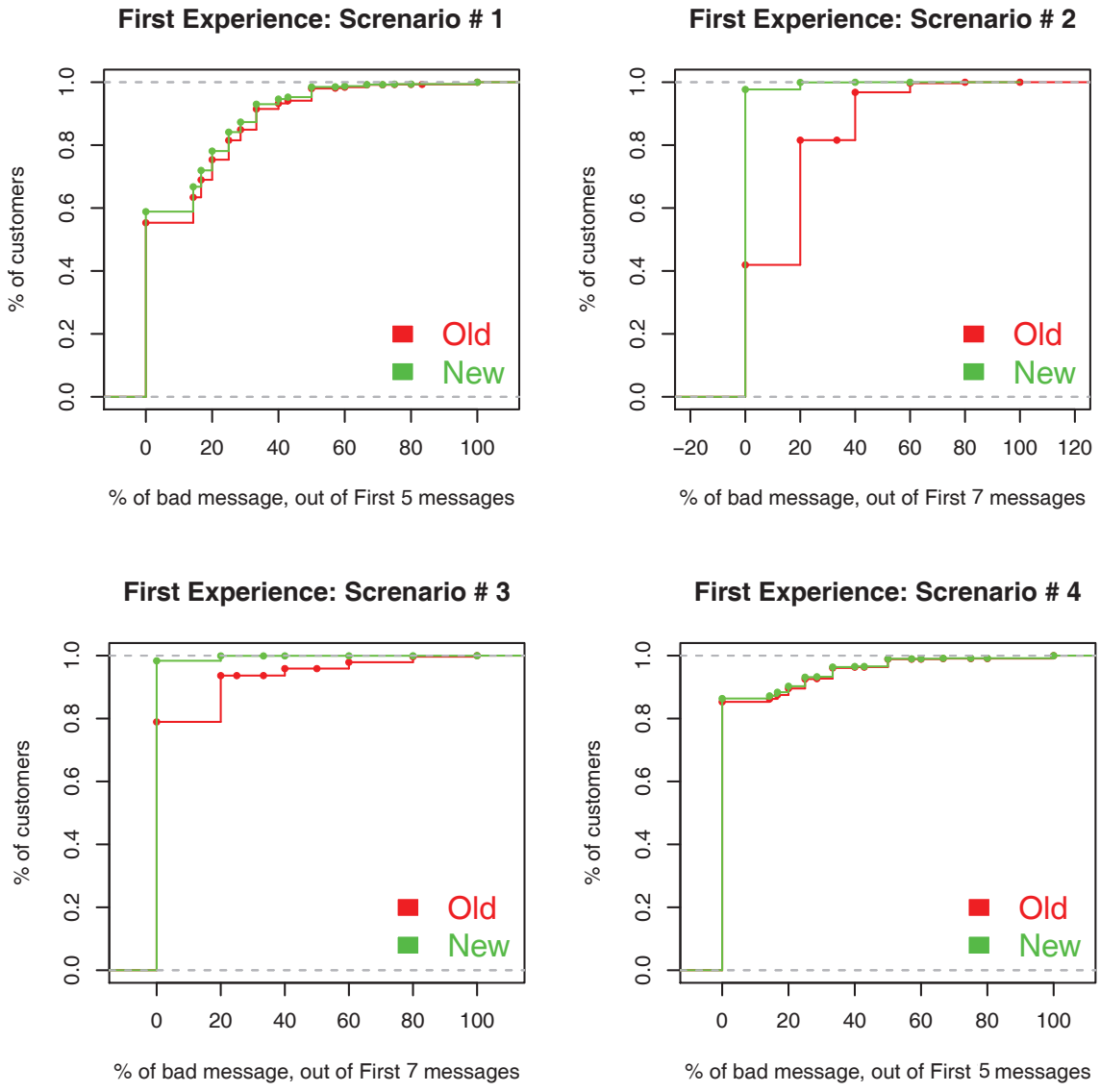

Figure 5.3: Enterprise System 1 - User-centric initial (first) experience

Table 5.3: DS2 - scenario-centric Comparison of performance

Scenario #	Mean	Median	% of bad instances
Login	545.31	531.25	13.11
Browse	517.05	515.63	12.87
Purchase	6062.84	5984.38	20.8

response time for the first "n" instances of a scenario for each user. The idea behind a metric for initial experience is to see how many users had a bad experience initially. In this paper, we measure bad experience or bad instance of a scenario as an instance with the following delay or response time criteria:

$delay > median(scenario) + standard\ deviation(scenario)$

This threshold uses the median and standard deviation of response time for each scenario.

Out of the "n" initial instances, we measured the percentage of bad instances. We choose the value of "n" to be the maximum of the 10th percentile and 5. For example, in Figure 5.1, we can see the histogram of the total number of instances of a user for each scenario in ES 1. For scenario 1 and 4, the 10th percentile was 1 and using only the first instance to measure the initial experience would give us only two levels of initial experience, either 0% or 100%. Hence, for these two scenarios, we considered the first 5 instances to measure the initial experience and for scenarios 2 and 3, we considered the first 7 instances (as the 10th percentile value was 7). In ES 2, we considered n=11,11,9,5,5,5 for six scenarios and in DS2, n=5,7,5 for the 3 scenarios.

For the scenario-centric performance analysis, we do not need to define "n", since performance trend curves (discussed in subsection 5.2.2) over time can show the

scenario-centric initial performance.

Overall Experience: For the user-centric performance measure of overall experience, "n"' represents all instances for a user during the whole period of analysis. For example, if a user had 35 instances during one hour of analysis, we identified the number of bad instances in these 35 instances using the same threshold $delay > median(scenario) + standard\ deviation(scenario)$, then calculated the percentage of bad instances of that user.

For the scenario-centric performance measure of overall experience, we considered the mean, median and density of the response time, as well as the percentage of bad instances across all users. For example, if a scenario had 50 users and they had 500 instances of a scenario during one hour of analysis, we calculated the mean, median and standard deviation of response time for these 500 instances across all users. The mean value gives us the average across all observation data including outliers, while the median quantifies the central point of the dataset, neutralizing the outliers. We graphically compare the distribution of response time with kernel density plots (regular and cumulative).

With median and standard deviation values, we used the same threshold $delay > median(scenario) + standard\ deviation(scenario)$ to define a bad instance. Finally, we measured the scenario-centric % of bad instances in each scenario (shown in Table 5.2 for ES 1 and in Table 5.3 for DS2). For scenario-centric comparison of mean response time values between old and new ES1 version, we also used Welch's statistical t-test to check whether the mean difference is statistically significant.

Findings - Enterprise System 1: The user-centric performance analysis showed major performance improvements for 2 (out of 4) use case scenarios (scenario #2

and #3 in Figure 5.2) while the scenario-centric analysis showed major performance improvements for 1 use case scenario (scenario #3 in Table 5.2).

Figure 5.2 shows the differences (between old and new version) in user-specific response time for enterprise system 1 as a cumulative density plot. This figure looks similar to the Initial experience curves for the users in Figure 5.3.

In scenario #2, using median values (scenario-centric, across all users) we find that most of the instances in both versions had the same response time of 406 units indicating no performance deviation between these two versions. Using mean values, we find approximately 3.72% performance degradation (statistically significant) on average in the newer version, while from user-centric analysis, (Figure 5.2) we found that the newer version of the system performs better than the older version. We found that more than 90% of the users using the older system had at least 1 bad instance (% of bad instances > 0 in Figure 5.2, scenario #2, marked with red arrow), while less than 5% of the users had such a bad experience in the new system (marked with green arrow).

In scenario #3, we can see from Table 5.2 that on average and for most of the instances (median), the newer version had a better overall performance (5.73% better on average, statistically significant and 30.43% using median). The user-specific overall experience curve for this scenario also confirms that performance improvement (Figure 5.2).

Scenario-centric analysis (shown in Table 5.2) shows that, depending on the scenario, the mean difference between the old and new version was within the range of 0.39% to 16.27%.

Findings - Enterprise System 2: In ES 2, user-centric analysis in 5 (out of 6)

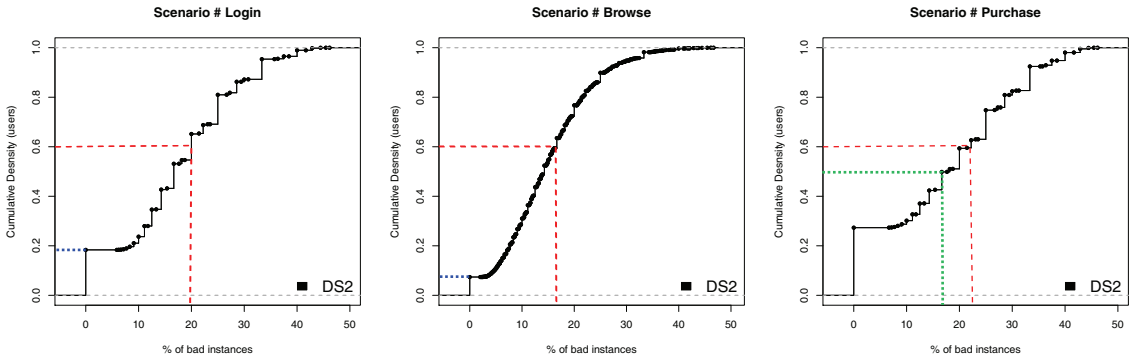

Figure 5.4: DS2-Cumulative Density Plot of user Overall Experience in % of bad instances

scenarios showed that 50% of the users in each scenario had a bad performance in at least 20% of their instances, while the scenario-centric analysis showed 2.09% to 18.35% bad instances across all users. Different perspectives of the system performance are shown by these two approaches. We do not show the data and curves for ES 2 as these looked similar to the ones shown for ES 1.

Findings - DS2: In DS2, a user-centric analysis for the 3 (out of 3) scenarios showed that 40% (100% - 60%) of the users in each scenario had a bad performance at least in 15% of their instances (red dashed lines in Figure 5.4), while scenario-centric analysis shows 13.11% to 20.8% bad instances across all users (Table 5.3). Having more than 15% bad instances for 40% of the customers may be too high from a user-centric perspective of the system performance.

Figure 5.4 shows the cumulative density function (CDF) plot of the user's overall experience (user-centric analysis result) and Table 5.3 shows the mean, median and % of bad instances across all users (scenario-centric analysis result) .

We can see that on average, a login or browse scenario in DS2 takes around 500

milliseconds or 0.5 seconds and around 12-13% of instances had a response time higher than median + standard deviation. From a scenario-centric perspective, the performance may look good, but when we look at the user-centric view in Figure 5.4 (blue dotted lines), we find that more than 80% of the users using DS2 had at least one performance deviation. Moreover, we found around 5-10% of the users with more than 30% of the requests having a performance deviation.

Response time (Table 5.3) of the purchase scenario show that on average across users, the response time was more than 6 seconds (which is surprisingly high) and similar to the login and browse scenarios, 50% of the users perceived a bad performance in more than 15% of their instances (green dotted line in Figure 5.4).

User-centric performance analysis shows users with serious performance problems in 10 scenarios (out of 13). These problems would have gone unnoticed if we had only performed a scenario-centric performance analysis.

5.2.2 How does the user and scenario-centric performance evolve over time?

Motivation: Progressive degradation of software performance is a common phenomenon that is called "software aging" [63]. To counteract software aging, operators measure, analyze and predict system response time (delay) and resource usages over time [34]. Software system performance trends can show how the system's performance changes over time while the system is running. Typically, aggregated summaries of the data (e.g., per minute or week) are plotted over time for system performance analysis. Again, the user aspect is missing in such time trends. A software tester may be interested in knowing how system performance changes for an

individual user over time. For example, it may be possible that in the beginning, the average response time for a user may be higher than later on (because of the initial registration process for each new user or due to a cache not being filled up yet), and the tester wants to investigate these. Moreover, it may be possible that based on scenario-centric performance trends (mean and median), a software system's performance is steady over time, while from the user-centric view, as he/she uses the system more and more, the performance actually degrades. The scenario and user-centric perspective of experience over time may have different performance trends.

Approach: From the scenario-centric perspective, the time trend of a system's performance is measured by collecting the instance response time and plotting this response time over the instance start/end time. For example, for the duration of our test analysis period, we collected the response time and instance start time for each scenario instance in the system. In a period of 1 hour for ES 1 and 10 hours for ES 2 and DS2, we used "loess" (locally weighted scatter plot smoothing) [25] to get the time trend curve for the system's performance.

For the user-centric time trend, for each scenario, we consider the first instance of each user as the user's time 1, the second instance as time 2, and so on. At each time X, we calculate the mean and median of the response time of all Xth instances. Since in most of the scenarios, only a few users (usually less than 20) have the highest number of scenario instances (for example in Figure 5.1 for ES 1), we drop data points with less than 100 users in our plots.

Findings - Enterprise System 1: In ES 1, 2 scenarios (out of 4) showed a different time trend when we considered the time from the scenario-centric and user-centric

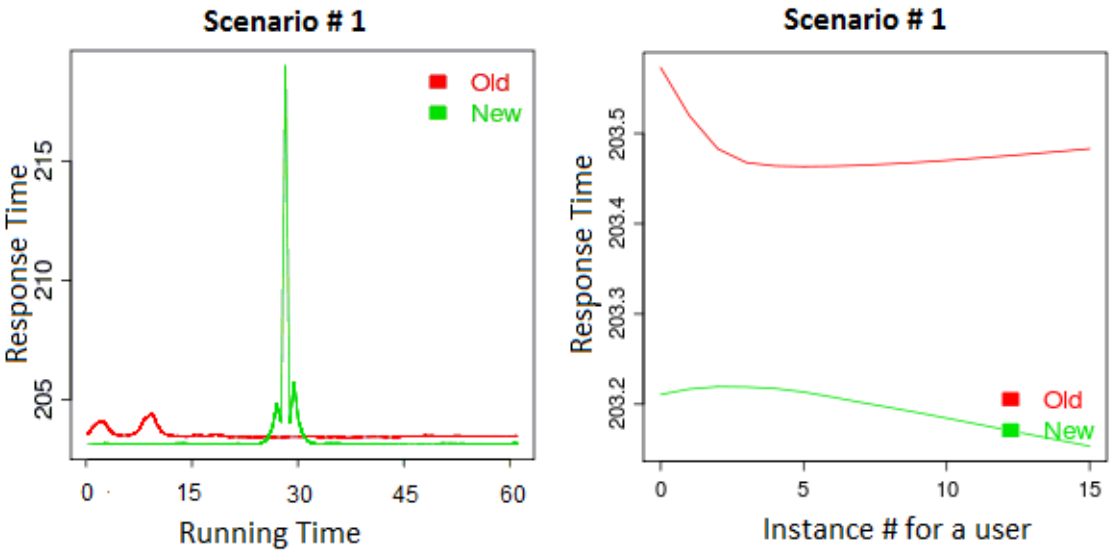

Figure 5.5: Scenario # 1 of Enterprise System 1 - performance trend over time from scenario-centric (left) and user-centric (right) view.

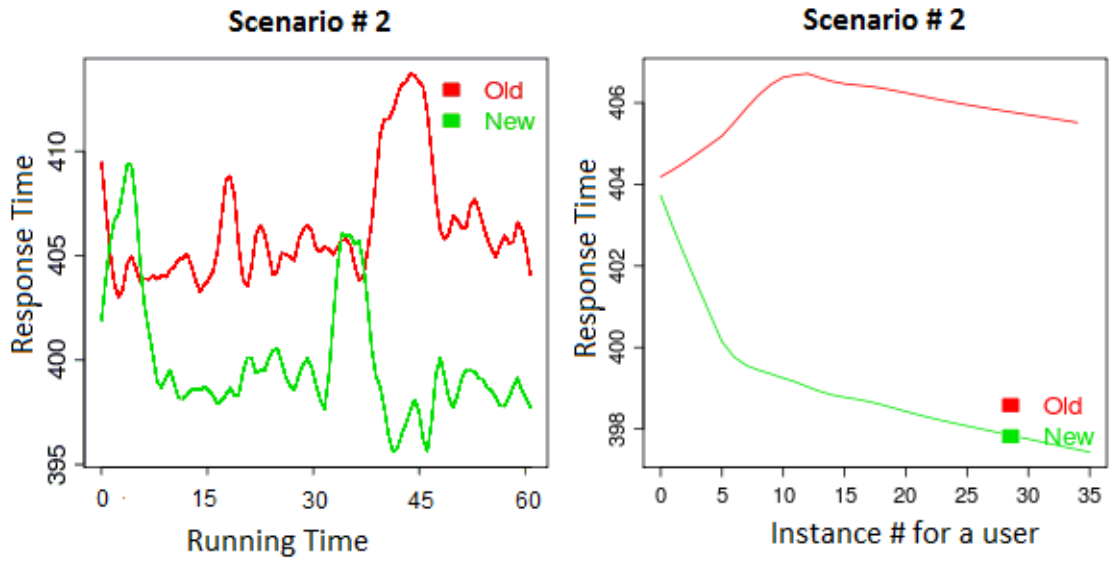

Figure 5.6: Scenario # 2 of Enterprise System 1 - performance trend over time from scenario-centric (left) and user-centric (right) view.

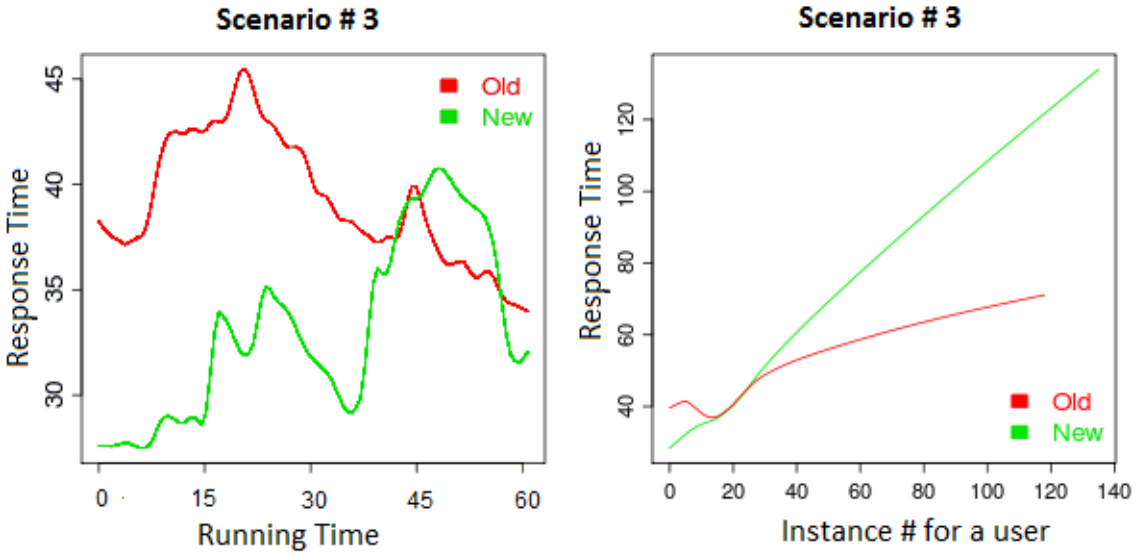

Figure 5.7: Scenario # 3 of Enterprise System 1 - performance trend over time from scenario-centric (left) and user-centric (right) view.

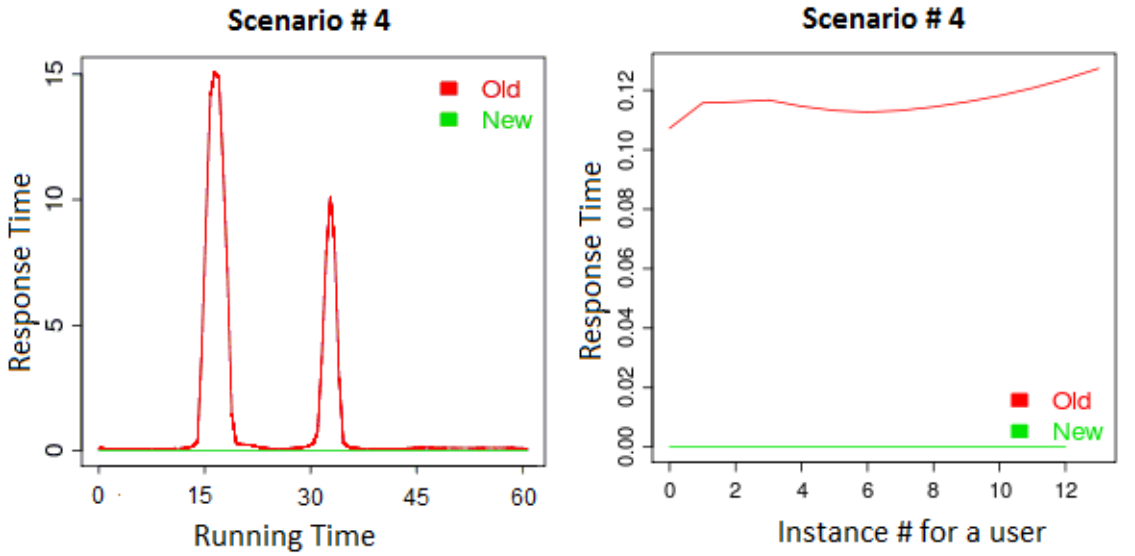

Figure 5.8: Scenario # 4 of Enterprise System 1 - performance trend over time from scenario-centric (left) and user-centric (right) view.

Table 5.4: Number of users and scenario instances in each Scenario.

Scenario # in ES 1 (old version)	# of scenario instances	# of users
1	142,528	29,769
2	498,838	30,000
3	2,025,645	128,104
4	391,959	123,462
Scenario # in ES 2	# of scenario instances	# of users
1	8,387	500
2	8,287	500
3	7,051	500
4	4,616	500
5	4,310	487
6	3,908	500
Scenario in DS2	# of scenario instances	# of users
Login	138,318	19,984
Browse	418,395	21,385
Purchase	139,733	21,384

perspectives. Both time trends showed useful information and one cannot be replaced by another.

Figure 5.5, 5.6, 5.7 and 5.8 show the performance trend over time comparison between scenario-centric and user-centric view for scenario #1, #2, #3 and #4 respectively.

The scenario-centric trend for all scenarios show that most of the time, the older version had a higher response time, although in the middle of the test run for scenario #1, the newer version had a very large response peak. From the scenario-centric trend lines, we can identify the time periods during the test data analysis hour when the

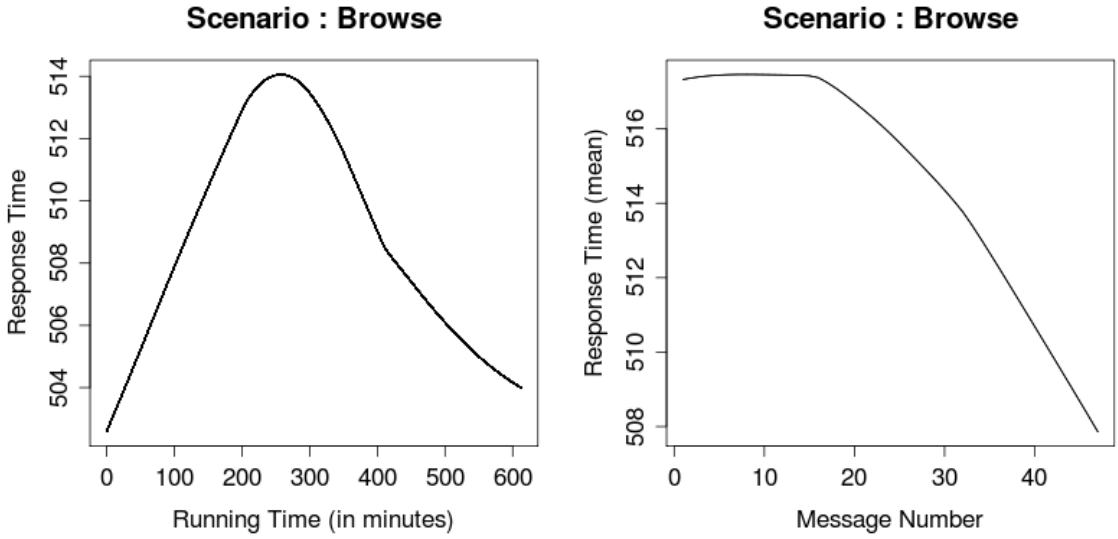

Figure 5.9: DS2 - Scenario-centric (left) and user-centric (right) time trend of system performance for "browse" response time

system performed poorly for each scenario and version.

From the user-centric performance trend lines, we can see that the older version had a higher response time (similar to the scenario-centric trend line), although we could not see any peak as we found in the scenario-centric trend lines of scenario #1.

Moreover, in scenario #3, in the user-centric view (Figure 5.7, right side) , the users who are involved in more than 30 instances (3,409 or 2.66% users), perceived a very bad performance (in comparison to the older version) for those instances after the 30th one. This information could not be observed in scenario-centric analysis (Figure 5.7, left side) .

In scenario #2 (Figure 5.6, right side), for the older version, the performance got worse too for users over time, but in the newer version (in green) this problem seems to be fixed. Such a conclusion cannot be made from the scenario # 1 and # 4 curves (right side figures of Figure 5.5 and 5.8) as the Y axis scale suggests that the

fluctuation was very small in comparison to the median value.

Findings - Enterprise System 2: In ES 2, 5 scenarios (out of 6) showed a completely different time trend when we considered the time from the scenario and user perspective. However, user-centric and scenario-centric trends were complementary both showing useful trends for performance analysis. From the user-centric trend lines, we could understand how the system performance changed over time for an individual user. However, from scenario-centric trend lines, we could only understand how the overall system performance evolved over time.

Findings - DS2: In DS2, 1 scenario (out of 3) showed a different time trend when we considered the time from scenario-centric and user-centric perspective. Figure 5.9 shows how the DS2 performance for the browse scenario changes over time. From the scenario-centric trend curve, we can see that the response time increased first, then decreased in this six hour period, while from the user's perspective, we found that as a user browses the DVD store more, his performance to the system improves. The other two scenarios, login and purchase showed similar performance trends in both scenario-centric and user-centric analysis. Please refer to Appendix B for the omitted graphs of Dell DVD Store test data analysis.

The time trends drawn from the scenario and user-centric perspective show different aspects in 8 scenarios (out of 13) in our case study.

5.2.3 How consistent are the user-centric and scenario-centric performance characteristics?

Motivation: Performance consistency is an important factor for a user. A user using a system regularly with a slow response time may get used to that slow performance,

but if the system performs sometimes very fast and sometimes very slowly, this would show to the user that a better experience is possible and he/she is likely to start demanding such level of services. Consistent poor performance may be more expected than inconsistent poor performance in the case of a network service provider that has some users with specific Quality of Service (QoS) requirements that specify that the user needs a consistent but poor performance (relative to the system's performance capability). Preference of consistency is dependent on the system operator's and the user's performance expectation.

Figure 5.10 shows the simplest possible combination of performance and consistency. For overall experience, a user may have either Good (G) or Bad (B) experience while for performance consistency, a user may have either Consistent (C) or Inconsistent (I) performance. The first square, which represents the users with consistently good experience perceived from the system, is desired by everyone (user and system operator). If it is not possible to have most of the users in square #1, the system operator, depending on his or her preference, may prefer to have most of the users in one of the other three squares. Intuitively, it seems that any user in the third block (marked as 3), is the most unlucky one. Depending on the QoS offered to the user (if the QoS offered is a slow but steady performance), this can also be the desired performance by the system operator. Research in the field of marketing shows that there may be some rational users who may want to rationally choose a lower quality [67]. For example, some users may be interested in a cheaper service that may not give them the best overall performance (i.e., response time), but still they would want this cheaper performance to be consistent. Such performance and consistency analysis (shown in Figure 5.10) can identify the users in each of these performance

regions.

Approach: To measure the performance consistency over time (scenario-centric), we calculated the median and standard deviation of scenario instances in every one minute interval and plotted the median ± standard deviation values in every such interval using "loess". For the performance consistency trend over user-perspective of time, we plotted the median ± standard deviation values for the 1st, 2nd, 3rd, ... nth scenario instances across all users (similar to research question 2). Median ± standard deviation plots create two lines forming the standard deviation band in which most of the scenario instance delay varies.

To measure the performance consistency, we used variance (which is the square of standard deviation). For scenario-centric variance, we calculated the variance of all instances across all users. For user-centric variance, we calculated the variance of scenario instance response time for each user. To observe the relation between user-centric performance consistency and the user's overall performance experience as described in Figure 5.10, we plotted the scatter-plot of variance vs overall performance for each user (Figure 5.13). In the same plot, we added the scenario-centric variance values by using horizontal lines.

Findings - Enterprise System 1: The performance consistency trends from user perspective and the overall performance vs consistency scatter plot cannot replace the scenario-centric analysis approach, but can be helpful to better assess the performance of a software system.

Figure 5.11 and 5.12 show the performance consistency trend comparison between scenario-centric (left side) and user-centric (right side) view for scenario #2 and #3 respectively. We used green for the new version and red for the older version to

show the area of standard deviation band. For scenario-centric view of scenario #2 (Figure 5.11, left side), we see that in the beginning (from X=0 to X=1000) the newer version performed less consistently than the older version (green region of newer version showing beyond the overlapped region). For scenario-centric view of scenario #3 (Figure 5.12, left side), we see that the older version performed less consistently than the newer version (wider error band for older version).

From the user-centric performance consistency trend (in Figure 5.11 and 5.12, right side), we see very inconsistent performance during initial scenario instances of the newer version. We are not showing the other two scenarios (#1 and #4) as they did not show any visibly significant difference between scenario-centric and user-centric performance consistency error bands. Both user-centric and scenario-centric consistency trends show useful information and the purpose of one cannot be achieved by the other one. Knowledge about performance consistency trends are useful for the user and/or the service provider to ensure that the system meets the QoS requirement consistently as per the agreement between them.

Figure 5.13 shows the variance and overall performance experience of each user as a single semi-transparent dot. In this Figure, darker (green or red) dots and dotted regions corresponding to over-plotting of several semi-transparent dots represent the existence of many users in that region. Clearly visible scattered green dots with higher values of variance in scenario #2 and #3 show many users in the newer version with performance inconsistencies. Moreover, most of the green dots (new version) to the left of the plot indicate the lower % of bad instances experienced by the users in the new version.

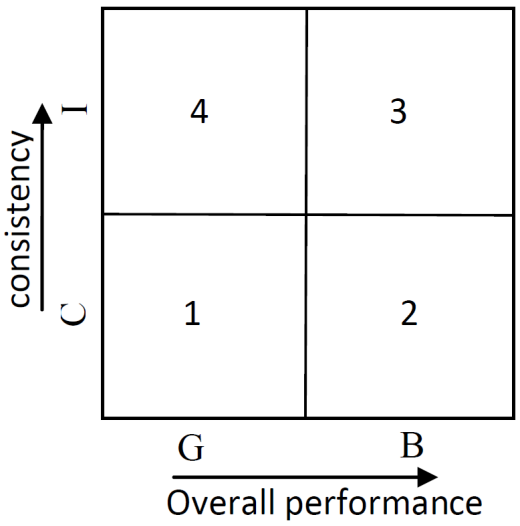

Figure 5.10: Four options to choose between consistency and overall performance.

Depending on the threshold set by the system operator to define good/bad performance and consistent/inconsistent performance, each plot in Figure 5.13 can be divided and mapped into the four regions as shown in Figure 5.10. For example, in scenario #2, most of the green dots along the Y axis indicate that most of the users in the newer version are either in square #1 or #4 (most of them had a good experience depending on the threshold). In scenario #3, most of the users using both versions were in either region #1 or #2 (most of them had a consistent experience depending on the threshold).

Findings - Enterprise System 2: The scenario-centric and user-centric time trend of performance inconsistency did not show any difference for ES2. 6 scenarios (out of 6) had a similar error band curve from the scenario and user-centric perspective. More investigation by variance vs % of bad instances showed that most of the users had a consistent performance, although they were scattered into different levels of performance experience (scattered along X axis).

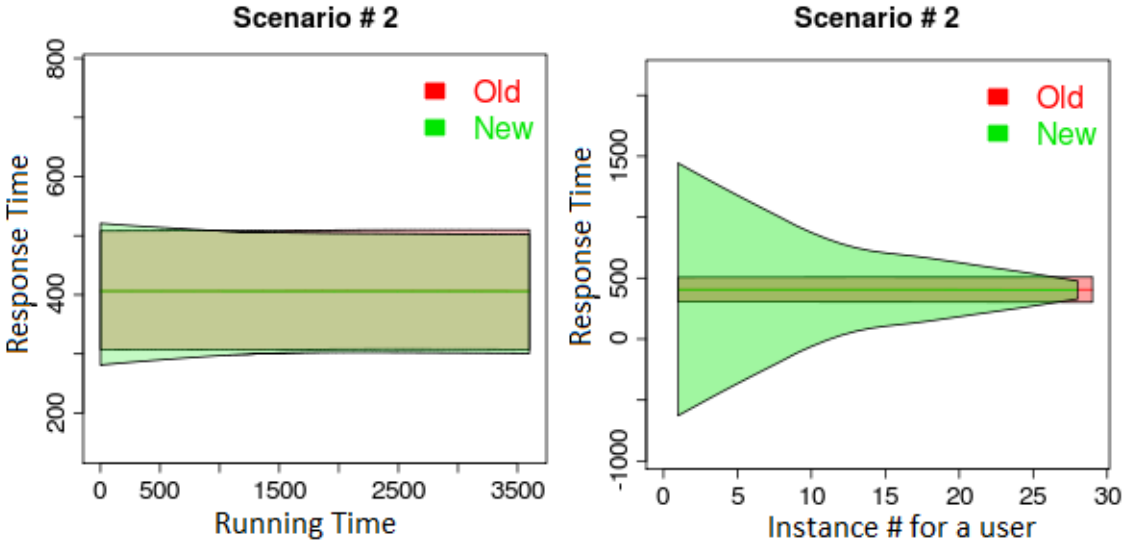

Figure 5.11: Scenario # 2 of Enterprise System 1 - scenario-centric (left) and user-centric(right) performance consistency - showing the error band after using LOWESS smoother.

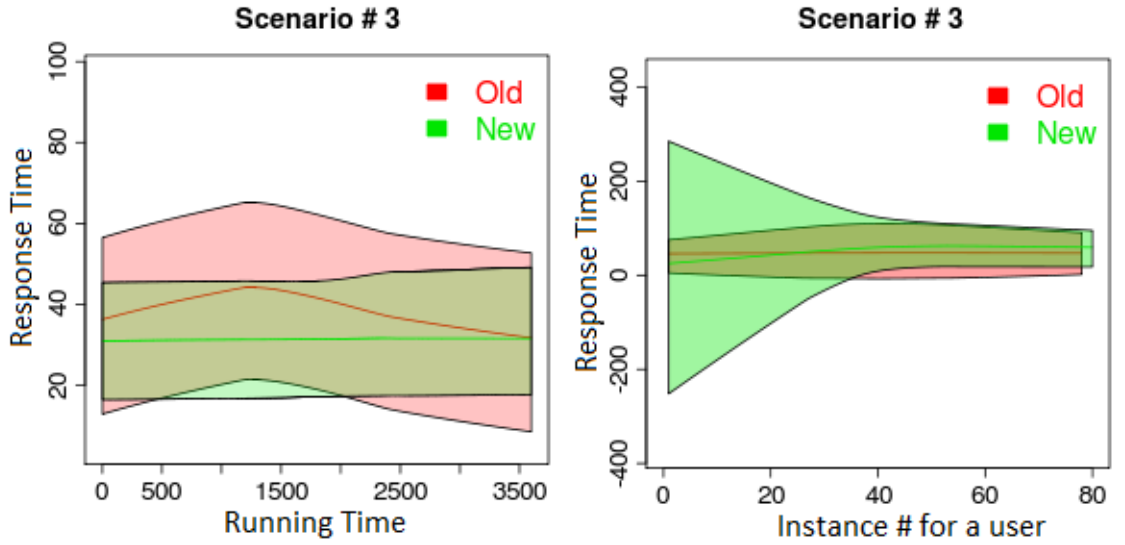

Figure 5.12: Scenario # 3 of Enterprise System 1 - scenario-centric (left) and user-centric(right) performance consistency - showing the error band after using LOWESS smoother.

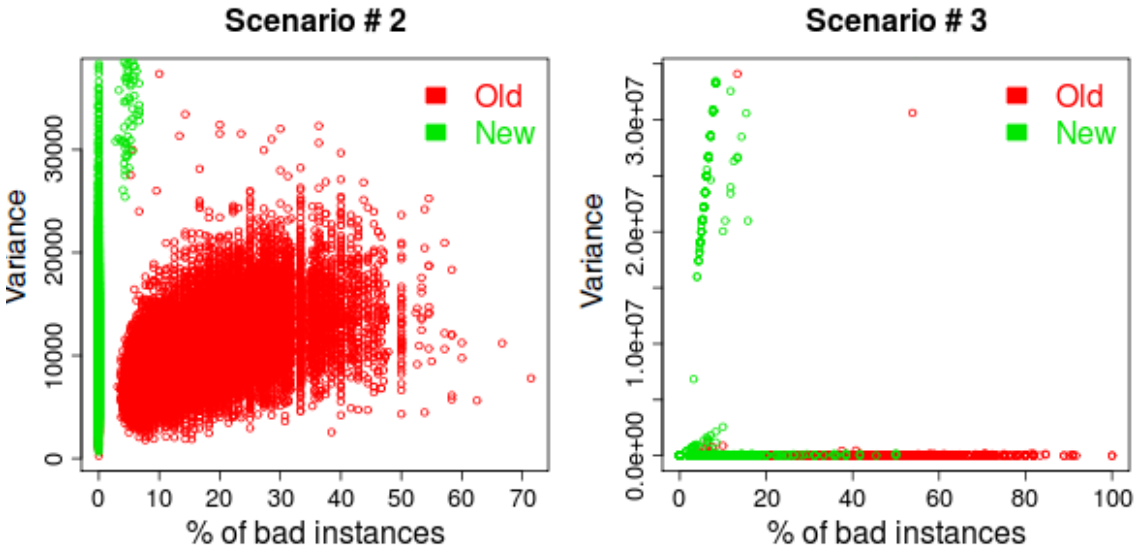

Figure 5.13: Enterprise System 1 - User's Experience Variance vs Overall Performance Experience scatter plot.

Findings - DS2: The scenario-centric consistency plot for DS2 (Figure B.8, B.9 and B.10 in Appendix B), showed us that the DS2 overall performance variance was consistent over time (two almost straight and parallel lines formed by median ± standard deviation lines). The user-centric performance consistency plot also showed a similar curve.

However, user-centric analysis of performance variance vs experience (% of bad scenario instance) in Figure 5.14 shows that for the login and browsing scenarios, the users faced less performance variance than for purchasing (relatively higher Y axis value of variance in the purchase scenario than login and browse scenario). In the purchase scenario, a straight dark line along the Y axis represents a large number of users with good but inconsistent performance experience (region #2 in Figure 5.10). Moreover, the many vertical long lines for the purchase scenario in Figure 5.14 show very inconsistent performance perceived by the users irrespective of their overall

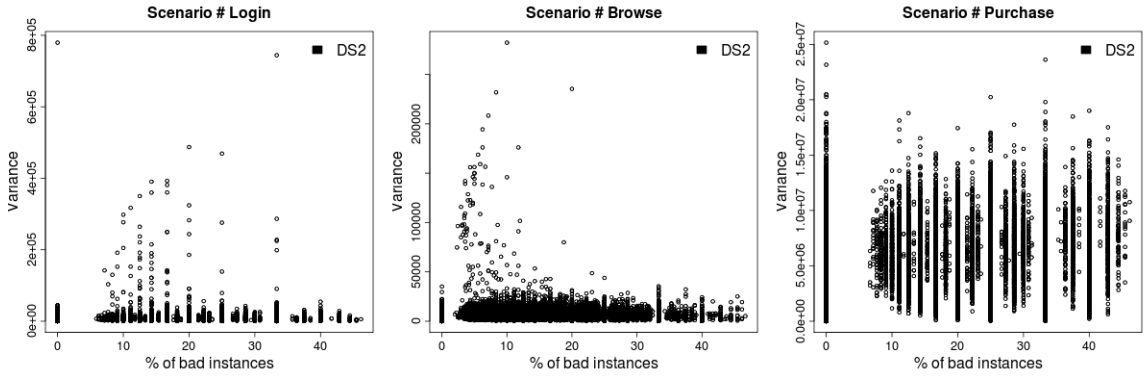

Figure 5.14: DS2 - User's Performance Experience Variance vs Overall Performance Experience scatter plot

performance.

User-centric analysis showed us a different trend from scenario-centric performance consistency trends.

5.3 Threats to Validity

Our study was done on three software systems of different scale (ultra large, large and small) and different type (open-source and commercial). To generalize our user-centric performance analysis results, more projects from different software domains should be studied.

This study was based on virtual users that are simulated by software load generators. Although no real users were involved in this study, the load generators used in test environments follow realistic loads, typically derived from data about real users.

We used a median ± standard deviation (1 standard deviation) threshold to define a scenario instance response time to be affected by bad performance. Depending on

the performance criteria of the system used, this threshold should be adjusted. However, we also checked the median ± 3 * standard deviation threshold and found similar findings (overall observation, although the percentages and values were different).

5.4 Chapter Summary

Our goal in this study was to verify the importance of user-centric software performance analysis. Researchers in many fields such as marketing, software usability, software reliability have a broad knowledge on how to do user-centric analysis in their field. By the use of these analysis techniques in software performance analysis, we were able to detect performance problems in a software system that could not be detected from existing scenario-centric performance analysis techniques. User-centric analysis could track the percentage of users with very bad experience and identify users with good but very inconsistent performance. However, we also found scenarios where performance peaks were detected by existing scenario-centric analysis, but not by user-centric analysis. Hence, we believe the user-centric analysis should be used as a complementary to the scenario-centric performance analysis approaches.

Chapter 6

Conclusion and Future Work

The studies presented throughout this thesis are summarized and our main findings are highlighted in this chapter. The limitations and possible directions for future work are also presented.

6.1 Summary

Maintaining software performance with changing system environments and user demand is essential for a software project to keep their users satisfied in today's competitive market.

Despite being an important software quality characteristics, we observe that performance related bugs are rarely studied independently in software maintenance research and practice. Possible reasons for such an observation may be the preconception that performance bugs are not much different than other types of bugs.

To identify if there is any difference between performance bug and non-performance bugs, and thus if a different treatment is warranted, we performed an empirical study on both performance and non-performance bugs of the Mozilla Firefox and Google Chrome web browsers to validate our research hypothesis:

Performance bugs have different characteristics than other bugs and should be treated differently in software maintenance research and practice.

In Chapter 3, we quantitatively studied the difference among three type of bugs, i.e., performance, security and others in Mozilla Firefox and Google Chrome project. Being considered to be very special and of interest, security bugs were studied to provide context for our findings. Our findings suggest that different types of bugs have different characteristics and hence quality assurance should take into account the specific bug type. For performance bugs, we found that they require more time to fix after being assigned, require more experienced developers and require more source code changes (in terms of LOC). These findings provide some evidence of the underlying differences in performance bugs or their fixing process.

To better understand this difference, we qualitatively studied performance bugs in the Mozilla Firefox and Google Chrome projects. We also studied non-performance bugs in this study to ensure that the findings we have for performance bugs are specific to performance bugs only. We found that performance bugs are often hard to reproduce. In Firefox, 19% of the performance bugs were fixed out of the blue without any concrete link to the patch or change that fixed them. Furthermore, in 8% of the performance bugs of Firefox, bug commenters became frustrated enough to switch to another browser, compared to only 1% for non-performance bugs. 34% of performance bugs in Firefox and 36% in Chrome provided concrete measurements of, e.g., CPU usage, disk I/O, memory usage and the time to perform an operation in their report or comments. In addition, twice (32%) as many performance bugs as non-performance bugs in Firefox provided at least one test case in their bug report or comment. 47% of the performance bugs in Firefox and 41% in Chrome required bug comments to

co-ordinate the bug analysis, while only 25% of the non-performance bugs required them. These findings show that performance bugs have different characteristics, in particular regarding the impact on users and developers, the context provided about them and the bug fix process.

In our further investigation in Chapter 5 based on one of our findings that performance bugs make users very frustrated, we dug more into user-centric performance analysis and found that introducing user-centric analysis for software performance would provide us with a complementary view over the commonly used scenario-centric analysis.

The above evidence provide support for our hypothesis, that, performance bugs have different characteristics and hence, software maintenance research and practice should consider them differently. Project managers, developers, testers, and researchers should consider adopting performance bug specific fixing process, tools and techniques. For example, based on our quantitative study findings, further work may be done to find out the reason behind the longest time to fix for performance bugs, and improve that time to fix performance bugs. Similarly, based on the qualitative analysis, further research is needed to improve the representation of the "steps to reproduce" for performance bug reports, and for developing more optimized means than collaborative discussions to identify the root cause of performance bugs.

6.2 Limitations and Future Work

The work presented in this thesis has several limitations. Limitations specific to our studies are described in the corresponding sections (Section 3.4, 4.5 and 5.3) of their respective chapters. In this section, we outline the overall limitations of this thesis

and how future work may be designed to address them.

- Our results may be biased by the studied projects. Initially, we studied Mozilla Firefox project data in our qualitative study described in Chapter 4. Replication of our analysis using different projects was necessary to verify our findings for other large scale software projects.

  Hence, we studied Google Chrome bug repository data. We found that performance bugs are tossed more frequently and have the lowest entropy (complexity) than security and other bugs in Chrome project. Three type of bugs (performance, security and others) showed a statistically significant difference for most of the characteristics. However, we could not find the rationale of any of these quantitative study findings of Google Chrome project using our qualitative study findings. Hence, we could not constitute a quantitative and qualitative study findings comparison table for Google Chrome project (as we did for Mozilla Firefox project in Table 4.4).

- Although security and performance bugs showed differences for different characteristics in both projects, the findings were opposite for most of the characteristics. For example, performance bugs are tossed more in the Chrome project, but security bugs are tossed more in the Firefox project. More studies should be done on different projects to investigate the reasoning for different behavior of same bug type in different projects.

- We found some bugs in both projects Firefox and Chrome projects (11 in Firefix and 1 in Chrome) that were both tagged as a security and performance problem.

In our quantitative study, we put these bugs in both the security and performance groups. More studies are needed to understand the interplay between these two important types of bugs.

- We studied only the bug data of two open-source web browser projects in our initial study on performance bugs. However, to validate the point that our study of performance bugs can be beneficial, we picked one of our findings to investigate more and use it toward suggesting an improvement of the software performance analysis process. In this study, we had the chance to use data from enterprise systems along with an open-source e-commerce system. Again, we used performance test data from three projects for this user-centric performance analysis study that may need more study on other projects from different domains (i.e., size, business model, complexity).

- We only investigated one of our findings in detail to improve the software performance maintenance effort. However, studies based on other findings may discover ways to improve other quality assurance processes related to performance bugs.

- Our thesis tries to establish the importance of considering performance bugs separately from other non-performance bugs. However, studies on other types of bugs, i.e., usability, functionality, and serviceability bugs may be beneficial for improving those software quality characteristics in software and thus improve the overall performance. Further bug-type specific research can be done on these types of bugs.

Bibliography

[1] Draft standard for ieee standard classification for software anomalies. *IEEE Unapproved Draft Std P1044/D00003, Feb 2009*, 2009.

[2] Mozilla foundation security advisory. `http://www.mozilla.org/security/announce/`, February 2011.

[3] Mozilla super-review policy. `http://www.mozilla.org/hacking/reviewers.html`, February 2012.

[4] Mark S. Ackerman and Christine Halverson. Considering an organization's memory. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, CSCW '98, pages 39–48, New York, NY, USA, 1998. ACM.

[5] Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In *Proceedings of the 4th International Conference on Software Engineering Advances*, ICSEA '09, pages 216–221, Washington, DC, USA, 2009. IEEE Computer Society.

[6] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

[7] Pierre F. Baldi, Cristina V. Lopes, Erik J. Linstead, and Sushil K. Bajracharya. A theory of aspects as latent topics. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 543–562, New York, NY, USA, 2008. ACM.

[8] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *Software Engineering, IEEE Transactions on*, 30(5):295 – 310, may 2004.

[9] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, pages 259–272, 2004.

[10] Steffen Becker. Dependability metrics. chapter Performance-related metrics in the ISO 9126 Standard, pages 204–206. Springer-Verlag, 2008.

[11] Dane Bertram, Amy Voida, Saul Greenberg, and Robert Walker. Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In *Proc. of the 2010 ACM conference on Computer supported cooperative work*, CSCW '10, pages 291–300, 2010.

[12] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann. Quality of bug reports in eclipse. In *Proc. of the*

2007 OOPSLA workshop on eclipse technology eXchange, eclipse '07, pages 21–25, 2007.

[13] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 308–318, 2008.

[14] Christian Bird, Alex Gourley, and Prem Devanbu. Detecting patch submission and acceptance in oss projects. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, MSR '07, pages 26–, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *Proceedings of the 3rd international workshop on Mining software repositories*, MSR '06, pages 137–143, New York, NY, USA, 2006. ACM.

[16] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality? an empirical case study of windows vista. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 518–528, Washington, DC, USA, 2009. IEEE Computer Society.

[17] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[18] Cynthia M. Calongne. Designing for web site usability. *Journal of Computing Sciences in Colleges*, 16:39–45, March 2001.

[19] M. Cataldo, A. Mockus, J.A. Roberts, and J.D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *Software Engineering, IEEE Transactions on*, 35(6):864 –878, 2009.

[20] P.K. Chang and H.L. Chong. Customer satisfaction and loyalty on service provided by malaysian telecommunication companies. In *Proceedings of the 3rd International Conference on Electrical Engineering and Informatics (ICEEI)*, pages 1 –6, july 2011.

[21] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 595 – 604, 2002.

[22] R.C. Cheung. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, SE-6(2):118 – 125, march 1980.

[23] I. Chowdhury and M. Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. In *Special Issue on "Security and Dependability Assurance of Software Architectures," Journal of Systems Architecture*, 2010.

[24] Sunita Chulani, P. Santhanam, Darrell Moore, Bob Leszkowicz, and Gary Davidson. Deriving a software quality view from customer satisfaction and service data. In *Proceedings of European Conference on Metrics and Measurement*, 2001.

[25] W. S. Cleveland, S. J. Devlin, and J. B. Wagenaar. Locally-weighted regression: An approach to regression analysis by local fitting. *Journal of The American Statistical Association*, 1988.

[26] Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. In *Proceedings of the 16th Working Conference on Reverse Engineering*, WCRE '09, pages 135–144, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[28] Dell Inc. Dell dvd store database test suite, Version 2.1.

[29] Sport England. Briefing Note: confidence intervals and statistical significance within the Active People Survey. unpublished.

[30] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17 –23, 2000.

[31] M. Gegick, P. Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *Proceedings of the 7th international workshop on Mining software repositories*, pages 11 –20, May 2010.

[32] J. D. Gould and C. Lewis. Human-computer interaction. chapter Designing for usability: Key principles and what designers think, pages 528–539. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[33] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653 –661, July 2000.

[34] M. Grottke, Lei Li, K. Vaidyanathan, and K.S. Trivedi. Analysis of software aging in a web server. *IEEE Transactions on Reliability*, 55(3):411 –420, sept. 2006.

[35] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. "not my bug!" and other reasons for software bug report reassignments. In *Proc. of the ACM 2011 conference on Computer supported cooperative work*, CSCW '11, pages 395–404, 2011.

[36] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[37] A. Hindle, M.W. Godfrey, and R.C. Holt. What's hot and what's not: Windowed developer topic analysis. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, ICSM '09, pages 339 –348, 2009.

[38] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.

[39] ISO 9126:2003. *Software engineering – Product quality*. ISO, Geneva, Switzerland, 2003.

[40] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 111–120, New York, NY, USA, 2009. ACM.

[41] Zhen Ming Jiang. Automated analysis of load test ing results. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 143–146, New York, NY, USA, 2010. ACM.

[42] Zhen Ming Jiang, A.E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 125 –134, sept. 2009.

[43] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *Proceedings of the 24th IEEE International Conference on Software Maintenance*, pages 307–316, 2008.

[44] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proc. of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 155–170, 2011.

[45] Graham Kalton. *Introduction to Survey Sampling*. Sage Publications, Inc, 1983.

[46] Sunghun Kim and E. James Whitehead, Jr. How long did it take to fix bugs? In *Proceedings of the 3rd international workshop on Mining software repositories*, MSR '06, pages 173–174, New York, NY, USA, 2006. ACM.

[47] Andrew J. Ko and Parmit K. Chilana. Design, discussion, and dissent in open bug reports. In *Proc. of the 2011 iConference*, iConference '11, pages 106–113, 2011.

[48] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49:230–243, March 2007.

[49] Henry H. Liu. *Software Performance and Scalability: a Quantitative Approach*. Wiley & Sons, Inc., 2009.

[50] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society.

[51] Lionel Marks, Ying Zou, and Ahmed E. Hassan. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 11:1–11:8, New York, NY, USA, 2011. ACM.

[52] Alan Merten and Daniel Teichroew. The impact of problem statement languages on evaluating and improving software performance. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part II*, AFIPS '72 (Fall, part II), pages 849–857, New York, NY, USA, 1972. ACM.

[53] A. Mockus and J.D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24rd international conference on Software engineering*, ICSE '02, pages 503 – 512, 2002.

[54] Audris Mockus and David Weiss. Interval quality: relating customer-perceived quality to process quality. In *Proceedings of the 30th International Conference on Software engineering*, ICSE '08, pages 723–732, 2008.

[55] Audris Mockus, Ping Zhang, and Paul Luo Li. Predictors of customer perceived software quality. In *Proceedings of the 27th International Conference on Software engineering, ICSE*, pages 225–233, 2005.

[56] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[57] MySQL AB. Mysql community server, 2011, Version 5.5.

[58] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 521–530, New York, NY, USA, 2008. ACM.

[59] Stephan Neuhaus and Thomas Zimmermann. Security trend analysis with cve topic models. In *Proceedings of the 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 111–120, Washington, DC, USA, 2010. IEEE Computer Society.

[60] Dor Nir, Shmuel Tyszberowicz, and Amiram Yehudai. Locating regression bugs. In *Hardware and Software: Verification and Testing*, volume 4899 of *Lecture Notes in Computer Science*, pages 218–234. 2008.

[61] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14:286–315, June 2009.

[62] Lucas D. Panjer. Predicting eclipse bug lifetimes. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, MSR '07, pages 29–, Washington, DC, USA, 2007. IEEE Computer Society.

[63] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software engineering*, ICSE, pages 279–287, 1994.

[64] Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 37–48, Washington, DC, USA, 2007. IEEE Computer Society.

[65] C. R. Reis and R. P. de Mattos Fortes. An overview of the software engineering process and tools in the mozilla project. In *Proceedings of the Open Source Software Development Workshop*, pages 155–175, 2002.

[66] Christian Del Rosso. Performance analysis framework for large software-intensive systems with a message passing paradigm. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 885–889, New York, NY, USA, 2005. ACM.

[67] R. T. Rust, J. J. Inman, J. Jia, and A. Zahorik. What you don't know about customer-perceived quality: The role of customer expectation distributions. *Marketing Science*, 18:77–92, 1999.

[68] Adrian Schröter. Msr challenge 2011: Eclipse, netbeans, firefox, and chrome. In *Proc. of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 227–229, 2011.

[69] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Ken-ichi Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Proceedings of the 17th Working Conference on Reverse Engineering*, WCRE '10, pages 249–258, Washington, DC, USA, 2010. IEEE Computer Society.

[70] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 315–317, New York, NY, USA, 2008. ACM.

[71] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30:1–5, May 2005.

[72] Michael Terry, Matthew Kay, Brad Van Vugt, Brandon Slack, and Terry Park. Ingimp: introducing instrumentation to an end-user open source application. In *Proceeding of the 26th annual SIGCHI Conference on Human factors in computing systems*, pages 607–616, 2008.

[73] The Apache Software Foundation. Tomcat, 2010, Version 5.5.

[74] Stephen W. Thomas, Bram Adams, Ahmed E. Hassan, and Dorothea Blostein. Validating the use of topic models for software evolution. In *Proceedings of the*

2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM '10, pages 55–64, Washington, DC, USA, 2010. IEEE Computer Society.

[75] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the 4th International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.

[76] Elaine J. Weyuker and Filippos I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Softw. Eng.*, 26(12):1147–1156, December 2000.

[77] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, Dec 1945.

[78] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs: a case study on firefox. In *Proc. of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 93–102, 2011.

[79] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[80] M.F. Zibran, F.Z. Eishita, and C.K. Roy. Useful, but usable? factors affecting the usability of apis. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 151 –155, oct. 2011.

[81] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models*

in Software Engineering, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.

Appendix A

LDA Outputs for Security and Performance Bugs

Table A.1: LDA topics for performance bugs (30 topics)

Topic #	**Words in the Topic**
1	object properti obj call function script var jsval global src trace scope slot arrai record jscontext stack abort jsobject
2	buffer stream data pars convert charset parser copi read token input write url length file utf cpp version call
3	cach page disk save data sourc entri view memori size store back disk_cach reload set histori user view_sourc session
4	font text select gfx editor type charact glyph size run metric gtk text_frame rang text_run line font_font textarea xft
5	style tabl css rule color border data cell row width context style_context sheet stop selector height style_data map hint
6	line byte line_byte event context layout metric shell metric_line line_layout layout_metric handl ref ptr entri img cach ref_ptr cach_entri

7	view paint rect frame widget window draw layer region clip updat invalid area manag render svg view_manag chang caret
8	window reproduc hang gecko build window_window problem result crash bug step user window_gecko report browser work step_reproduc click linux
9	java applet plugin sun jre sourc unknown unknown_sourc load version plug run sourc_sun consol java_applet thread netscap applet_applet oji
10	bug mark duplic bug_mark mark_duplic duplic_bug fix bug_bug build problem mac verifi netscap win work linux move keyword bugzilla
11	bookmark histori place menu browser select item queri search date filter visit result click bug remov moz compon match
12	scroll slow page window bug build problem cpu fix background mous testcas linux anim imag perform test move fast
13	dll plugin flash thread video xul xul_dll messag window call wait xpcom run process player base crash core plai
14	frame reflow line block state context layout block_frame htmlreflow frame_reflow ifram tabl pre box list reflow_state ipr_context htmlreflow_state metric
15	symbol window file load mode profil safe defer dll support system mod program mod_load time net_clr version clr updat
16	imag print gif img png decod gtk draw cairo gfx frame preview printer color src print_preview alpha size set

17	alloc arrai size memori arena malloc byte free pool sort stack number call string heap code patch bit lock
18	cpu tab window memori usag open problem bug close cpu_usag time comment run system ram page leak minut process
19	file pref startup servic profil load directori compon xul cooki read local jar instal manag zip browser user open
20	cpp src build line cpp_line base content base_src trunk seamonkei tinderbox html build_seamonkei file event build_tinderbox shell crash window
21	bug comment don work make problem time issu thing code fix repli repli_comment doesn set chang good point start
22	patch attach comment creat creat_attach fix test check comment_attach attach_patch code bug remov chang review call branch make updat
23	mail messag folder imap thunderbird attach file server msg log download problem account send inbox email compos open displai
24	event timer window call process queue event_event thread listen fire notif user messag docum handler pend flush load set
25	node elem content attribut docum tabl entri result context hash kei add call list set child index function parent
26	build compil gcc enabl code check optim version option linux defin disabl function add nspr gener bit platform instruct
27	page load html script testcas attach docum test creat file url content page_load creat_attach link browser render bug text

28	server connect request proxi network send socket ssl smtp data log thread tcp respons problem host state secur psm
29	time test perform profil run build number call faster sec improv result slower creat regress spent spe slow machin
30	lib thread usr usr_lib compon event lib_compon symbol app debug wait cpp system home version bin run modul servic

Table A.2: LDA topics for Security bugs (10 topics)

Topic #	**Words in the Topic**
1	event cpp line window dll src crash xul shell base view xul_dll handl bytes line_bytes widget pre cpp_xul app
2	regress obj patch var src root revision function test object branch regress_regress xml script cvsroot index call jsval assert
3	header cooki test content user uri server channel respons cach request site jar page css redirect url proxi host
4	crash build src tinderbox line build_tinderbox depend winnt cpp winnt_depend cpp_line framework xul version trunk window debug base depend_layout
5	frame line layout crash cpp block reflow assert state testcas pre frame_cpp gener list cpp_line block_frame style layout_gener content
6	patch branch test check testcas code trunk window repli don file make work set gecko verifi land releas doesn
7	imag size length frame decod data buffer charact alloc width crash lib overflow read usr_lib usr png memori state
8	window load url uri page file open content docum secur script html browser javascript href user locat link dialog
9	content docum elem node index xul child remov bind xbl row src crash tree cpp parent print arrai content_base
10	object window function script properti princip call scope global obj set wrapper xpconnect check proto access eval outer prototyp

Table A.3: LDA topics for security bugs (5 topics)

Topic #	**Words in the Topic**
1	patch object branch regress obj function test check src call code properti set revision repli var script root arrai
2	window patch test branch check content docum testcas load script secur work trunk code url gecko uri file make
3	patch test file user check repli code imag html header problem set branch data error type request issu don
4	frame crash patch testcas cpp branch line block layout pre content reflow assert state list check trunk rev build
5	line cpp crash event window src dll build xul base view shell tinderbox depend trunk build_tinderbox handl cpp_line frame

Appendix B

Dell DVD Store

In this Appendix, we provide the Dell DVD Store performance test configuration we used, and our finding graphs for any replication study.

Table B.1: Dell DVD store configuration

Database size	Minimum (10MB)
Number of threads	50
Ramp rate	10
Warm up time	1
Run Duration	11 hours
Customer think time	5
Percentage of new customers	1
Average number of searches per order	3
Average number of items returned in rach search	5
Averga number of items purchased per order	5
Web Application Type	JSP

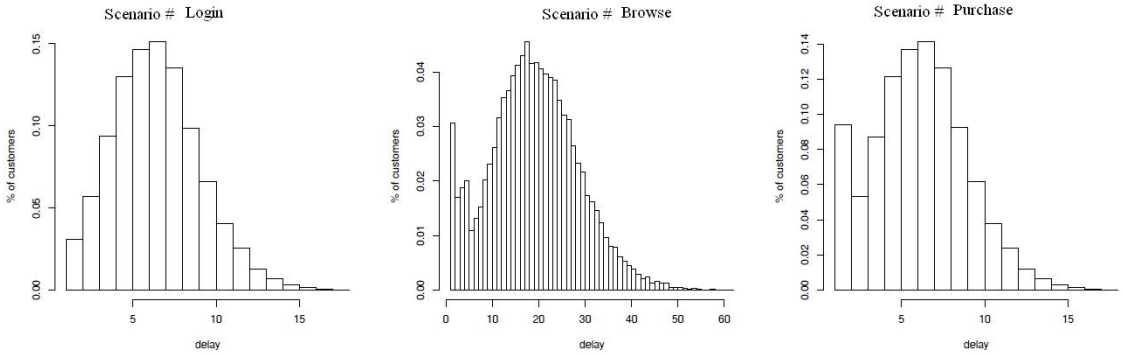

Figure B.1: Histogram of number of messages in DS2

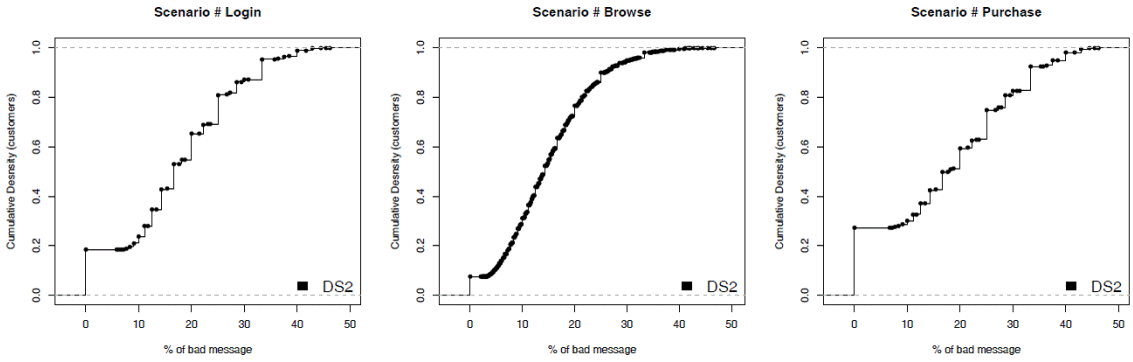

Figure B.2: DS2 user-centric overall performance (Cumulative Density plot for % of bad messages)

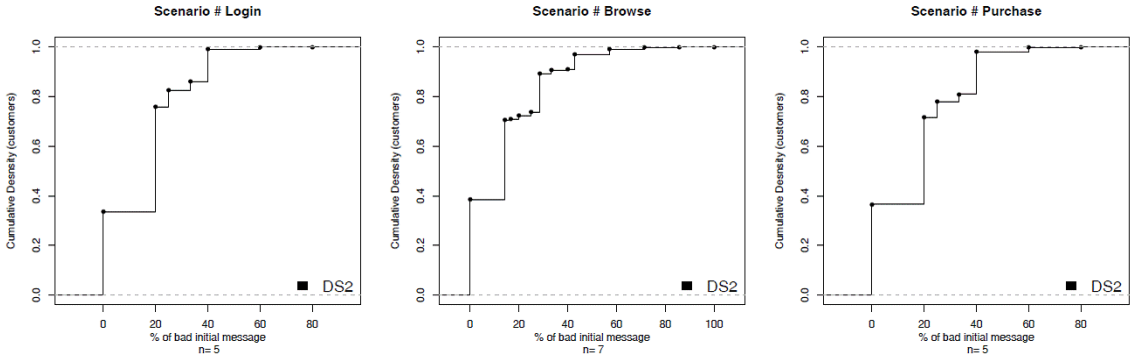

Figure B.3: DS2 - user-centric initial experience of customers (Cumulative Density for % of bad initial messages)

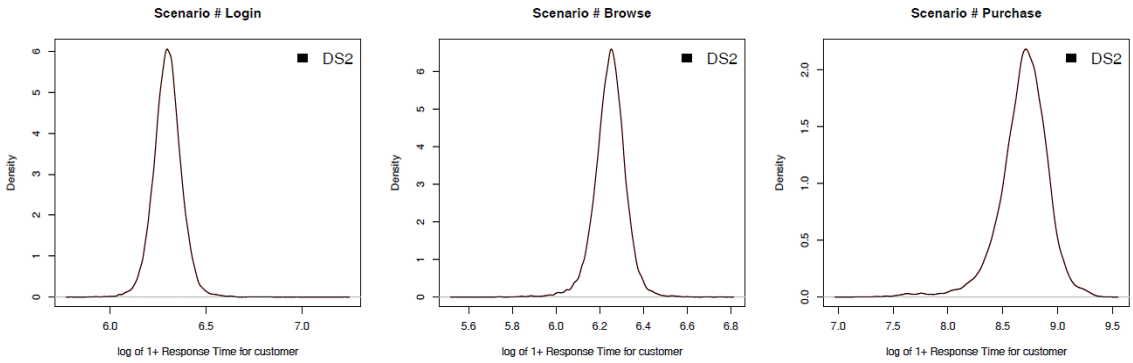

Figure B.4: Density of Response time for each scenario (DS2) Logged Scale

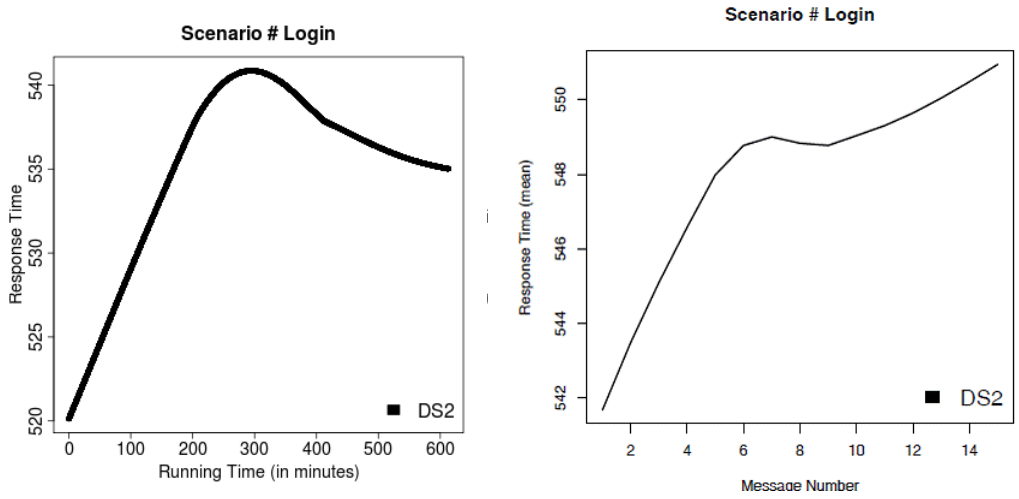

Figure B.5: DS2 - login scenario - comparison between scenario-centric (left) and user-centric (right) performance trend over time

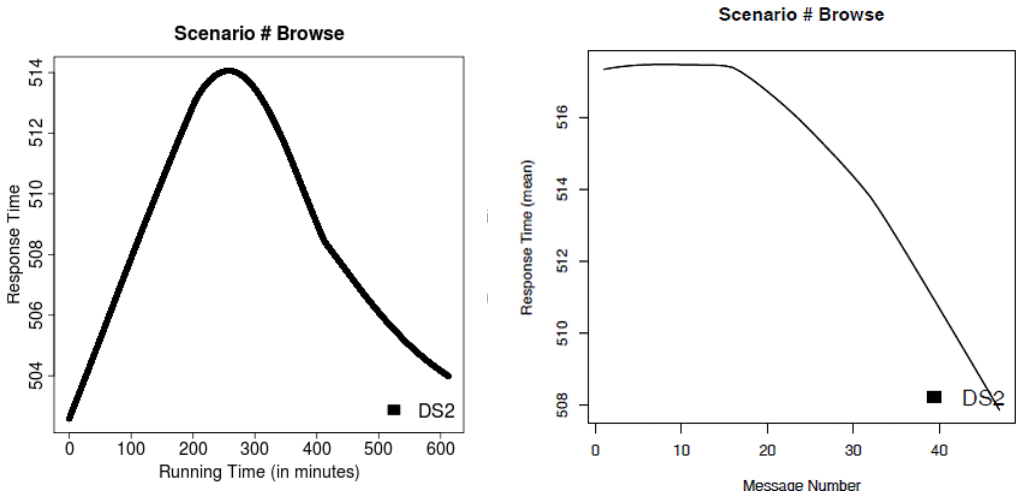

Figure B.6: DS2 - browse scenario - comparison between scenario-centric (left) and user-centric (right) performance trend over time

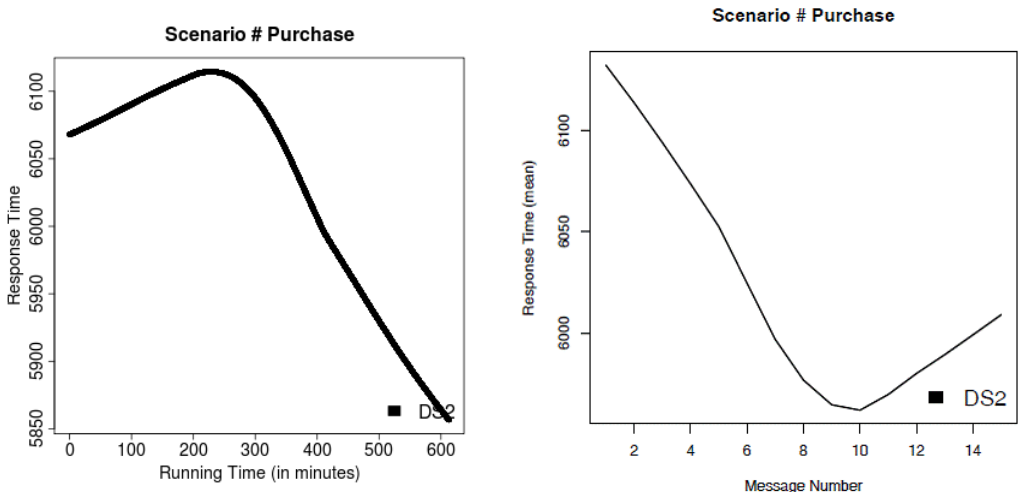

Figure B.7: DS2 - purchase scenario - comparison between scenario-centric (left) and user-centric (right) performance trend over time

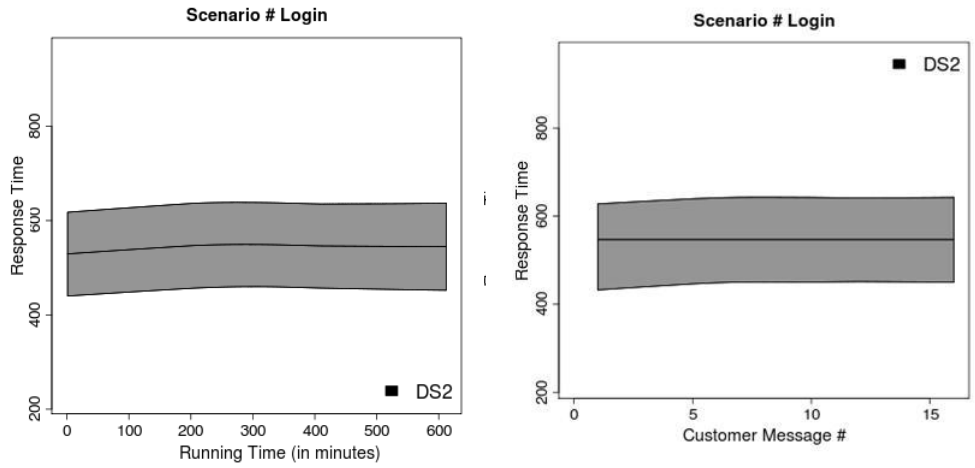

Figure B.8: DS2 - login scenario - comparison between scenario-centric (left) and user-centric (right) performance consistency over time

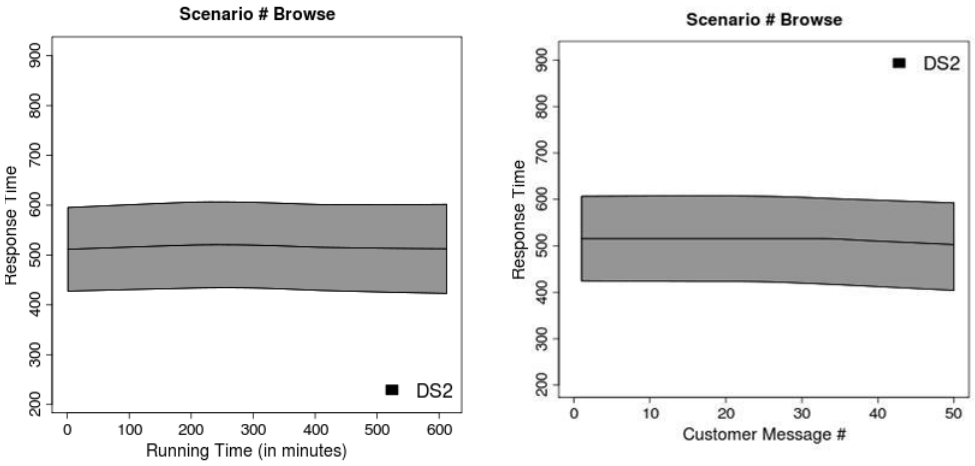

Figure B.9: DS2 - browse scenario - comparison between scenario-centric (left) and user-centric (right) performance consistency over time

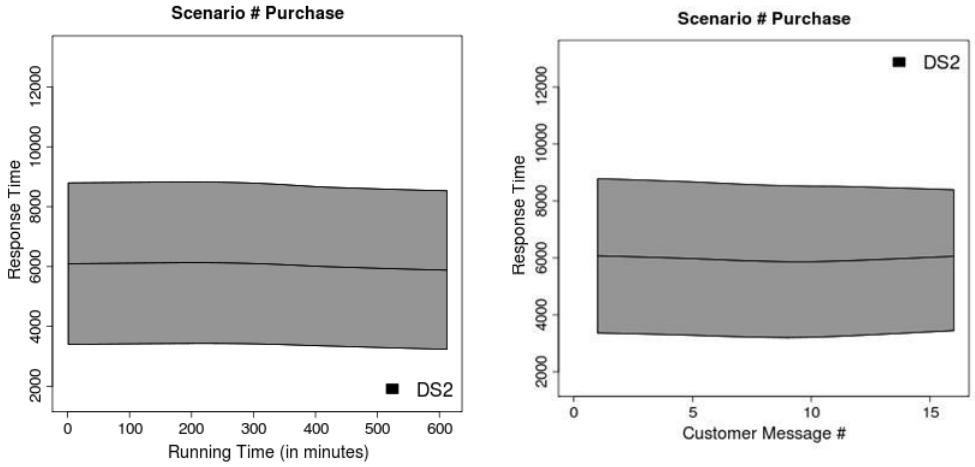

Figure B.10: DS2 - purchase scenario - comparison between scenario-centric (left) and user-centric (right) performance consistency over time

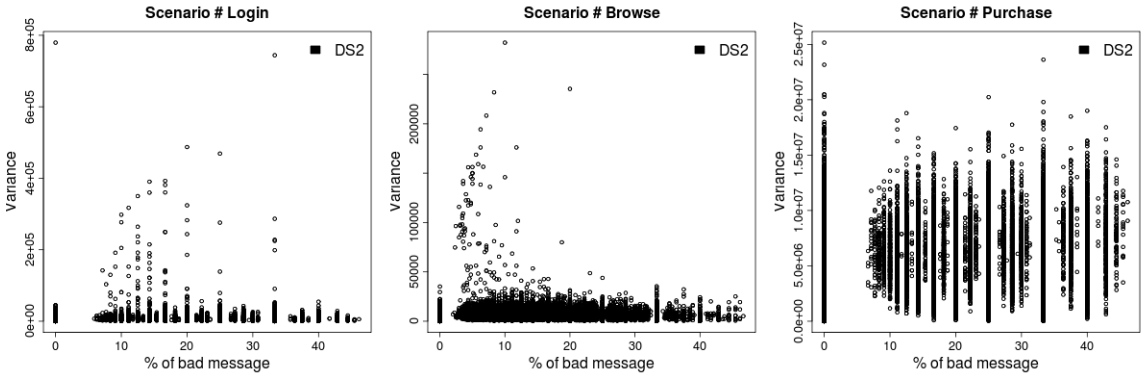

Figure B.11: User-centric Consistency vs overall performance for DS2