Understanding, Detecting, and Diagnosing Real-World Performance Bugs

by

Linhai Song

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 10/16/2015

The dissertation is approved by the following members of the Final Oral Committee:
Shan Lu, Associate Professor, Computer Science (at the University of Chicago)
Benjamin Liblit, Associate Professor, Computer Sciences
Aws Albarghouthi, Assistant Professor, Computer Sciences
Remzi Arpaci-Dusseau, Professor, Computer Sciences
Xinyu Zhang, Assistant Professor, Electrical and Computer Engineering
Darko Marinov, Associate Professor, Computer Science (at the University of Illinois at Urbana-Champaign)

To my parents, Zuoyu Song and Cuihua Fu.

Acknowledgments

First and foremost, I would like to express my wholehearted gratitude to my advisor, Professor Shan Lu, for her generous support and guidance during my PhD.

The first email I sent to Shan was to ask whether I could provide her name as my potential advisor when I applied for my student visa. Shan kindly approved this. I was lucky to begin my independent study with her immediately after I came to the U.S.. I can still remember the joy in my heart when she told me that she would pick me as her student after we submitted AFix.

Throughout my PhD, Shan gave me numerous invaluable academic advice, ranging from the details of how to present data in my paper to the broad vision of how to explore an initial research idea. Besides research, Shan also taught me how to better communicate with other people, how to manage my time, and how to be confident. Shan has served as my role model for the past five years. I am fortunate to have been able to observe and learn how she obtains new knowledge, conducts research, and succeeds in her career. The PhD journey is tough. I feel extremely thankful for Shan's patience, support, and actionable suggestions at each stage. I do not think I can reach the destination without her encouragement.

I gratefully thank Professor Ben Liblit and Professor Darko Marinov for their kind help during and after our collaboration. Ben and Darko are always encouraging and helpful whenever I encounter research problems. Ben also helped me go through all the paperwork and payment after Shan left to UChicago. I worked in Illinois for around one week, and it was an exciting experience to learn technical skills closely from Darko.

I would like to thank all the other committee members, Professor Remzi Arpaci-Dusseau, Professor Aws Albarghouthi, and Professor Xinyu Zhang, for their priceless comments and constructive criticism on my thesis. It is really my honor to have all these professors on my committee. I would also thank Professor Loris D'Antoni for coming to my defense and leaving invaluable feedback.

Thanks also go to every student in Shan's group and fellow students I worked with for the opportunity to learn together. I worked with Guoliang Jin on my first two research projects, and I want to thank him for helping me through my junior years. I want to thank Adrian Nistor for treating me to so many lunches and solving so many of the weird problems I encountered in Illinois. I want to thank Wei Zhang, Po-Chun Chang, Xiaoming Shi, Dongdong Deng, Rui Gu, Joy Arulraj, Joel Sherpelz, Haopeng Liu, and Yuxi Chen for making our group fun and exciting.

I also want to thank many other people at UWisconsin. Thanks to Professor David Page, Professor Michael Swift, Professor Susan Horwitz, Professor Jerry Zhu, and Professor Eric Bach for the knowledge I gained from their courses. Thanks to Angela Thorp for helping me through various administration stuff. Thanks to Peter Ohmann for suggestions about how to polish my talk. Thanks to my great roommates, Hao Wang, Ce Zhang, Jia Xu, and Huan Wang, for giving me such great company. Thanks to all the other good friends, Lanyue Lu, Wenfei Wu, Suli Yang, Yupu Zhang, Yiying Zhang, Jie Liu, Wentao Wu, Wenbin Fang, Weiyan Wang, Xiaozhu Meng, Junming Xu, Yimin Tan, Yizheng Chen, Ji Liu, Jun He, and Ao Ma for their help and support in different ways.

I benefit a lot from my summer internship at NEC Labs America. I want to thank the company as well as my mentor, Dr. Min Feng, for providing a terrific internship experience.

Last but not least, I want to thank my parents back in China for their unconditional love and trust. Whenever I struggle with my PhD life, my parents are always supportive and encouraging. My work would not be nearly as meaningful without them. No words can express my gratitude and love to them. I dedicate the whole thesis to the two most important persons in my life.

Contents

List of Tables

List of Figures

Nomenclature

1. *Performance bug:* performance bugs are software implementation mistakes that can cause inefficient execution. We will use "performance bugs" and "performance problems" interchangeably in this thesis, following previous work in this area [49, 75].

2. *Functional bug:* functional bugs are software defects that lead to functional misbehavior, such as incorrect outputs, crashes, and hangs. Functional bugs include semantic bugs, memory bugs, and concurrency bugs [54].

3. *Efficiency rule:* efficiency rules are principles inside programs, violations of which will lead to inefficient execution. Efficiency rules usually contain two components, which indicate where and how a particular code transformation can be conducted to improve performance while preserving original functionality.

4. *Root-cause information:* for a particular failure, root-cause information includes a static code region causing the failure, an explanation of why the failure happens, and potential fix suggestions.

5. *Inefficient loop:* inefficient loops are loops that conduct inefficient computation because of performance bugs.

6. *Diagnosis latency:* diagnosis latency is the time it takes to figure out the root cause of a failure. It is measured by how many failure runs are needed to conduct failure diagnosis.

7. *Runtime overhead:* runtime overhead is measured by comparing normal execution with monitored execution.

Abstract

Everyone wants software to run fast. Slow and inefficient software can easily frustrate end users and cause economic loss. The software-inefficiency problem has already caused several highly publicized failures. One major source of software's slowness is performance bug. Performance bugs are software implementation mistakes that can cause inefficient execution. Performance bugs cannot be optimized away by state-of-practice compilers. Many of them escape from in-house testing and manifest in front of end users, causing severe performance degradation and huge energy waste in the field. Performance bugs are becoming more critical, with the increasing complexity of modern software and workload, the meager increases of single-core hardware performance, and pressing energy concerns. It is urgent to combat performance bugs.

This thesis works on three directions to fight performance bugs: performance bug understanding, rule-based performance-bug detection, and performance failure diagnosis.

Building better tools requires a better understanding of performance bugs. To improve the understanding of performance bugs, we randomly sample 110 real-world performance bugs from five large open-source software suites (Apache, Chrome, GCC, Mozilla, and MySQL) and conduct the first empirical study on performance bugs. Our study is mainly performed to understand the common root causes of performance bugs, how performance bugs are introduced, how to expose them, and how to fix them. Important findings include that there are dominating root causes and fix strategies for performance bugs and root causes are highly correlated with fix strategies; that workload and API issues are two major reasons causing performance bugs to be introduced;

and that performance bugs require inputs with both special features and large scales to be exposed effectively. Our empirical study can guide future research on performance bugs, and it has already inspired our own performance-bug detection and performance failure diagnosis projects.

Rule-based bug detection is widely used to detect functional bugs and security vulnerabilities. Inspired by our empirical study, we hypothesize that there are also statically checkable efficiency-related rules for performance bugs, the violation of which will lead to inefficient execution. These rules can be used to detect previously unknown performance bugs. To test our hypothesis, we manually examine fixed performance bugs, extract efficiency rules from performance bugs' patches, and implement static checkers to detect rule violations. Our checkers find 332 previously unknown performance bugs. Some of found bugs have already been confirmed and fixed by developers. Our results demonstrate that rule-based performance-bug detection is a promising direction.

Effectively diagnosing user-reported performance bugs is another key aspect of fighting performance bugs. Statistical debugging is one of the most effective failure diagnosis techniques designed for functional bugs. We explore the feasibility and design spaces to apply statistical debugging to performance failure diagnosis. We find that statistical debugging is a natural fit for diagnosing performance problems, which are often observed through comparison-based approaches, and reported together with both good and bad inputs, statistical debugging can effectively identify coarse-grained root causes for performance bugs under the right types of design points, and one special nature of performance bugs allows sampling to lower the overhead of runtime performance diagnosis without extending the diagnosis latency.

Performance bugs caused by inefficient loops account for two-thirds of user-reported performance bugs in our study. For them, coarse-grained root-cause information is not enough. To solve this problem, we first conduct an empirical study to understand the fine-grained root causes of inefficient loops in the real world. We then design LDoctor, which is a series of static-dynamic hybrid analysis routines that can help identify accurate fine-grained root-cause information. Sampling is leveraged to further lower diagnosis overhead

without hurting diagnosis accuracy or latency. The evaluation results show that LDoctor can cover most root-cause categories with good accuracy and small runtime overhead.

Our bug-detection technique and performance failure diagnosis techniques, guided by our empirical study, complement each other to significantly improve software performance.

Chapter 1

Introduction

We all want software to run fast and efficiently. Software performance severely affects the usability of software systems, and it is one of the most important problems in computer science research. Performance bugs are one major source of software's slowness and inefficiency. Performance bugs are software implementation mistakes that can cause inefficient execution. Due to their non fail-stop symptoms, performance bugs are easy to escape from in-house testing and are difficult to diagnose. Nowadays, the urgency to address performance bugs is becoming even more important with new hardware and software trends and increasing concerns about energy constraints.

Facing the challenge of performance bugs, this dissertation proposes effective performance-bug detection and performance-bug diagnosis approaches based on a comprehensive characteristics study of real-world performance bugs.

1.1 Motivation

Slow and inefficient software can easily frustrate users and cause financial losses. Although researchers have devoted decades to transparently improving software performance, *performance bugs* continue to pervasively degrade performance and waste computation resources in the field [67]. Meanwhile, current support for combating performance bugs is preliminary due to the poor understanding of real-world performance bugs.

Following the convention of developers and researchers on this topic [13, 49, 67, 89], we refer to performance bugs as software defects that can slow down the

```
//GCC27733 & Patch
//expmed.c

struct alg_hash_entry
{
-    unsigned int t;
+    unsigned HOST_WIDE_INT t;
}

void synth_mult(...unsigned HOST_WIDE_INT t, ...)
{
   hash_index = t ...;
   if (alg_hash[hash_index].t == t ...)
   {
      ...
      return ;
   }
   ...
   //recursive computation
}
```

Figure 1.1: A real-world performance bug in GCC (The '-' and '+' demonstrate the patch.)

program. Relatively simple *source-code* changes can fix performance bugs and significantly speed up software while preserving functionality. These defects can**not** be optimized away by state-of-practice compilers, thus bothering end users.

Figure 1.1 shows an example of a real-world performance bug. A small mistake in the type declaration of hash-table entry `alg_hash_entry` causes the designed memoization for `synth_mult` not to work when `t` is larger than the maximum value that type-`int` can represent. As a result, under certain workloads, `synth_mult` conducts a lot of redundant computation for the same `t` values repeatedly, leading to 50 times slowdown.

Performance bugs exist widely in released software. For example, Mozilla developers have fixed 5–60 performance bugs reported by users *every month* over the past 10 years. The prevalence of performance bugs is inevitable because little work has been done to help developers avoid performance-related mistakes. In addition, performance testing mainly relies on ineffective

black-box random testing and manual input design, which allows the majority of performance bugs to escape [67].

Performance bugs lead to reduced throughput, increased latency, and wasted resources in the field. In the past, they have caused several highly publicized failures, causing hundred-million dollar software projects to be abandoned [68, 83].

Worse still, performance problems are costly to diagnose due to their non fail-stop symptoms. Software companies may need several months of effort by experts to find a couple of performance bugs that cause a few hundred-millisecond delay in the 99th percentile latency of their service [88].

The following trends will make the performance-bug problem more critical in the future:

Hardware: For many years, Moore's law ensured that hardware would make software faster over time with no software development effort. In the multi-core era, when each core is unlikely to become faster, performance bugs are particularly harmful.

Software: The increasing complexity of software systems and rapidly changing workloads provide new opportunities for performance waste and new challenges in diagnosis [22]. Facing the increasing pressure on productivity, developers cannot combat performance bugs without automated tool support.

Energy efficiency: Increasing energy costs provide a powerful economic argument for avoiding performance bugs. When one is willing to sacrifice service quality to reduce energy consumption [7, 58], ignoring performance bugs is unforgivable. For example, by fixing bugs that have doubled the execution time, one may potentially halve the carbon footprint of buying and operating computers.

Performance bugs may not have been reported as often as functional bugs, because they do not cause fail-stop failures. However, considering the preliminary support for combating performance bugs, it is time to pay more attention to them, as we enter a new resource-constrained computing world.

Techniques in this thesis	Functional Bug	Performance Bug
Rule-based bug detection	✓	✓
Statistical debugging	✓	✓
Sampling-based approach	✓	✓✓
LDoctor	✗	✓

Table 1.1: Techniques employed in this thesis.

1.2 Thesis Philosophy

The topic of this thesis is to provide better tool support to combat performance bugs. The philosophy of this thesis is to investigate existing approaches originally designed for functional bugs and try to apply, adapt, and extend them for performance bugs.

This philosophy is promising, because many advanced techniques have been proposed for functional bugs in each stage of software development. For example, many detection tools [14, 27, 38, 55, 69] can help point out previously unknown functional bugs, automated testing tools [11, 13] can exercise software under different setting or inputs and expose functional problems, and there are also diagnosis [43, 56, 57] and fixing tools [29, 42, 44] for reported functional bugs.

There are also uncertainties behind this philosophy. For example, to build effective performance-bug detection techniques, we need to know the common root causes of performance bugs, which are not obvious, due to the poor understanding of real-world performance bugs. For example, many functional failure diagnosis techniques highly depend on failure symptoms, such as crashes. Unfortunately, performance bugs have non fail-stop symptoms, and it is even unclear how to identify failure runs for performance bugs.

The following techniques are explored and extended for performance bugs in this thesis. All of these techniques were previously designed for functional bugs.

Rule-base bug detection: There are many correctness rules inside software. For instance, a lock has to be followed by an unlock, opened files must be closed, and allocated memory must be freed. Violations of these rules may lead to

critical system failures. Many techniques [14, 18, 27, 38, 55, 69] have been designed to detect rule violations and identify previously unknown bugs. In this thesis, we find that there are also statically checkable efficiency-related rules in software, and violating them would lead to inefficient computation. These rules can be leveraged to detect previously unknown performance bugs, as shown by the first row in Table 1.1.

Statistical debugging: Statistical debugging [3, 5, 43, 45, 56, 57, 86] is one of the most effective failure diagnosis techniques. It usually works in two steps. First, a set of runtime events, referred to as predicates, is collected from both success runs and failure runs. Second, statistical models are used to identify predicates that are highly correlated with failure. In this thesis, we find that after selecting the right types of predicates and statistical models, statistical debugging is effective for providing coarse-grained root-cause information for performance bugs, as shown by the second row in Table 1.1. This finding also motivates us to develop a tool, **LDoctor**, dedicated to identifying fine-grained root causes for performance bugs, as shown by the fourth row in Table 1.1.

Sampling-based approach: Sampling-based techniques [5, 43, 56, 57] collect predicates from production runs in a low overhead. Since less information is collected in one single run, sampling-based approaches usually rely on multiple runs to achieve the same diagnosis capability, sacrificing diagnosis latency. In this thesis, we find that sampling can keep the capability of statistical performance diagnosis while lowering the runtime overhead. Since root cause related predicates usually appear multiple times in one performance failure run, sampling works better for performance bugs than functional bugs, as shown by the third row in Table 1.1.

1.3 Dissertation Contribution

This dissertation works on three directions to address performance-bug problems: real-world performance-bug understanding, performance-bug detection, and performance failure diagnosis. The components in this dissertation interact and complement each other, as shown in Figure 1.2.

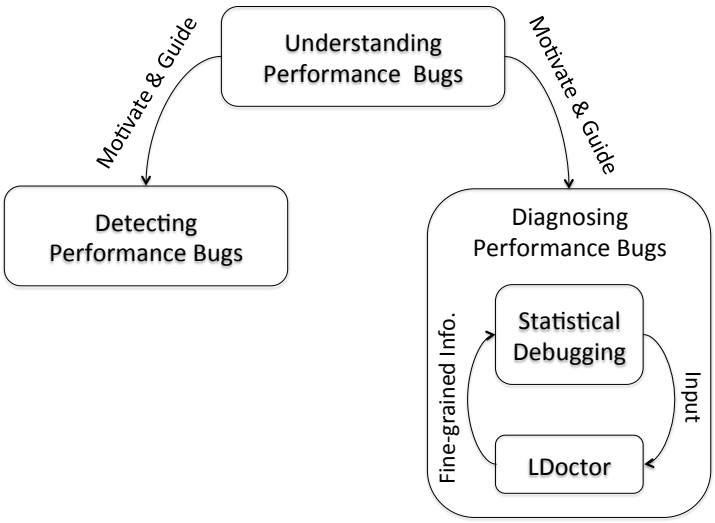

Figure 1.2: Interactions among the four components in this dissertation

1.3.1 Understanding performance bugs

Addressing performance-bug problems requires approaches from different aspects, and all these aspects will benefit from a better understanding of performance bugs.

This dissertation conducts a comprehensive characteristics study on a large number of real-world performance bugs collected from five widely used large open-source applications: Apache, Chrome, GCC, Mozilla, and MySQL [41]. This study reveals many interesting findings about performance bugs' root-cause patterns, how performance bugs are introduced, performance bugs' manifestation conditions, and fix strategies. It directly motivates the performance-bug detection work and performance-bug diagnosis work in this thesis.

1.3.2 Rule-based performance-bug detection

Guided by our characteristics study, we hypothesize that (1) efficiency-related rules exist; (2) we can extract rules from performance-bug patches; and (3) we can use the extracted rules to discover previously unknown performance bugs. To test these hypotheses, we collect rules from 25 Apache, Mozilla, and MySQL bug patches and build static checkers to look for violations of these rules [41].

Our checkers find 332 previously unknown Potential Performance Problems (PPPs) in the latest versions of Apache, Mozilla, and MySQL. These include 219 PPPs found by checking an application using rules extracted from a different application. Our thorough code reviews and unit testings confirm that each PPP runs significantly slower than its functionality-preserving alternate suggested by the checker. We report some of found PPPs to developers. 77 PPPs have already been confirmed by developers, and 15 PPPs have been fixed.

The main contribution of our bug-detection work is that it confirms the existence and value of efficiency rules: efficiency rules in our study are usually violated at more than one place, by more than one developer, and sometimes in more than one program. Rule-based performance-bug detection is a promising direction.

1.3.3 Statistical debugging for real-world performance bugs

Statistical debugging is one of the most effective failure diagnosis techniques proposed for functional bugs. We explore whether it is possible to apply statistical debugging to performance bugs and how to apply statistical debugging to performance bugs [90].

We first conduct an empirical study to understand how performance problems are observed and reported by real-world users. Our study shows that statistical debugging is a natural fit for diagnosing performance bugs, which are often observed through comparison-based approaches and reported together with both good and bad inputs. We then thoroughly investigate different design points in statistical debugging, including three different types of predicates and two different statistical models, to understand which design

point works the best for performance diagnosis. Finally, we study how one unique nature of performance bugs allows sampling techniques to lower the overhead of runtime performance diagnosis without extending the diagnosis latency.

1.3.4 Performance diagnosis for inefficient loops

Statistical debugging can accurately identify control-flow constructs, such as branches or loops, that are most correlated with the performance problem by comparing problematic runs and regular runs. Unfortunately, for loop-related performance bugs, which contribute to two-thirds of real-world user-perceived performance bugs in our study, this coarse-grained root-cause information is not enough. Although statistical debugging can identify the root-cause loop, it does not provide any information regarding why the loop is inefficient and hence is not very helpful in fixing the performance bug.

In order to provide fine-grained root-cause information, we design LDoctor, which is a series of static-dynamic hybrid analysis, for inefficient loops. We build LDoctor through two steps. The first step is to figure out a taxonomy of the root causes for common inefficient loops. The second step is to follow the taxonomy and design analysis for each root-cause subcategory. To balance accuracy and performance, we hybridize static and dynamic analysis. We further use sampling to lower the runtime overhead. Evaluation using real-world performance bugs shows that LDoctor can provide good coverage and accuracy with small runtime overhead.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes previous work on characteristics study of performance bugs, performance-bug detection, performance failure diagnosis, and other related topics. Chapter 3 presents our characteristics study of real-world performance bugs. Chapter 4 focuses on our rule-based performance-bug detection work. Chapter 5 explains our work on exploring how to apply statistical debugging to performance failure diagnosis. Chapter 6 discusses LDoctor, which can effectively diagnose inefficient loops.

Chapter 2

Background and Previous Work

This chapter discusses previous work on performance bugs related to this thesis, including empirical studies on performance bugs (Section 2.1), performance-bug detection (Section 2.2), performance failure diagnosis (Section 2.3), and other techniques to fight performance bugs (Section 2.4).

2.1 Empirical Studies on Performance Bugs

Recently, several empirical studies have been conducted on real-world performance bugs. They all have different focuses.

Zaman et al. [107] compare the qualitative difference between performance bugs and non-performance bugs across impact, context, fix and fix validation through studying 400 bugs from Mozilla Firefox and Google Chrome. Their study is conducted on different aspects of performance bugs from our study. For example, in their fix study, they focus on how many discussions occur among developers, whether the bug depends on other bugs, and whether the reporter provides some hints or patches during reporting. In our fix study, we focus on the fix strategies used in the final patches.

Liu et al. [60] randomly collect 70 performance bugs from smartphone applications to study the characteristics of these bugs. Similar to our study, they also find that there are common patterns for performance bugs and that these patterns can be used to detect previously unknown bugs. Since their performance bugs are collected from smartphone applications, they have some findings unique to mobile performance bugs. For example, the three most common types of performance bugs in their study are GUI lagging, energy

leak, and memory bloat. They also have some findings that conflict with those of our study. For example, they find that inputs with a small scale are sufficient to expose performance bugs.

The empirical study conducted by Nistor et al. [74] is similar to our bug characteristics study in Chapter 5 in that it also finds that performance problems take a long time to get diagnosed and the help from profilers is very limited. However, the similarity ends here. Different from our study in Chapter 3, this work does not investigate the root causes of performance bugs. Different from our study in Chapter 5, this work does not study how performance problems are observed and reported by end users. Its bug set includes many problems that are not perceived by end users and are instead discovered through developers' code inspection.

Huang et al. [39] study 100 performance regression issues to understand what types of changes are more likely to introduce performance regressions. Similar to our study, they find that performance regressions are more likely to happen inside a loop and that many performance regressions involve invoking expensive function calls. Unlike our work, their study focuses on performance regression.

2.2 Performance-bug Detection

Many performance-bug detection tools have been proposed recently. Each aims to find a specific type of hidden performance bugs before the bugs lead to performance problems observed by end users.

Some tools [22, 25, 100, 103] detect runtime bloat, a common performance problem in object-oriented applications. Some tools [101, 104] detect unnecessary references leading to memory leak. Xu et al. [102] target low-utility data structures with unbalanced costs and benefits. Cachetor [72] detects cacheable computation through dynamic dependence profiling and value profiling. A series of abstracts is proposed to reduce the detection overhead. Xu [98] detects reusable objects, data structures, or data contained in the data structures. In a follow-up work [99], a new object lifetime profiling technique is proposed to improve the reusability detection results. Chen et al. [12] detect database-related performance anti-patterns, such as fetching excessive data from database and

issuing queries that could have been aggregated. WAIT [2] focuses on bugs that block the application from making progress. Liu and Berger [59] build two tools to attack the false sharing problem in multi-threaded software.

Similar to our detection work in Chapter 4, Liu et al. [60] extract two rules from studied performance bugs, and detect violations to report unknown performance bugs. The bug patterns they can detect are lengthy operations in main thread and frequently invoked heavy-weight callback. Unlike our work, their bug detection technique focuses on performance bugs inside Android applications.

There are techniques targeting inefficient loops. Toddler [75] detects inefficient nested loops by monitoring memory read instructions. Given a nested loop, read value sequences from the same instructions inside the inner loop are compared across different iterations of the outer loop, and the nested loops will be reported as inefficiency when repetitive value sequences are found. Caramel [73] presents four static checkers to detect performance bugs with Cond-Break fixes. Every detected bug is associated with a loop and a condition. When the condition becomes true, the loop will not generate any results visible after the loop terminates. Therefore, the loop can be sped up by adding an extra break statement. Xiao et al. [97] profile programs to calculate complexity for each basic block, and identify a loop as an input-dependent loop, if the loop can cause complexity change between outside the loop and inside the loop. CLARITY [76] can statically detect asymptotic performance bugs caused by traversing a data structure repeatedly.

These tools are all useful in improving software performance, but they target different types of performance bugs from the performance-bug detection technique (Chapter 4) in this thesis. With different design goals, these performance-bug detection tools are not guided by any specific performance symptoms. Consequently, they take different coverage-accuracy trade-offs from performance diagnosis techniques (Chapter 6). Performance diagnosis techniques try to cover a wide variety of root-cause categories and are more aggressive in identifying root-cause categories, because they are only applied to a small number of places that are known to be highly correlated with the specific performance symptom under study. Performance-bug detection tools

have to be more conservative and try hard to lower false positive rates, because they need to apply their analysis to the whole program.

2.3 Performance Failure Diagnosis

Diagnosis tools aim to identify root causes and possibly suggest fix strategies when software failures happen.

2.3.1 Profilers

Profilers are widely used by developers to diagnose performance problems. For example, both gprof [1] and oprofile [77] are widely used profilers. To use gprof, programs must be compiled by using "-pg" option. For each call instance, gprof will record its caller. For each static function, gprof will record how many times it is called. At run time, gprof will look at program counter for every 0.01 second and record which function is being executed right now. gprof provides two types of profiling results: flat profile, which will rank executed functions based on how much time is spent on them, and call graph, which shows how much time is spent on each function and its children. Oprofile does not instrument programs. Oprofile asks users to specify hardware events and sampling frequency. Every time the specified number of events occur, oprofile will record the value of program counter and call stack. Similar to gprof, oprofile can also provide flat profile and call-graph information.

Research progress has also been made on profilers. Both AlgoProf [108] and aprof [16, 17] try to associate complexity information to bottleneck functions. The difference between AlgoProf and aprof is that AlgoProf considers the size of recursive data structure as the input size, while aprof estimates the input size by using the amount of distinct memory first accessed by a routine or its descendants with memory read. Calling Context Tree (CCT) is a compact representation of calling contexts encountered during program execution. Each node on a CCT corresponds a routine, and the path from the root to the node represents a unique calling context. Zhuang et al. [110] optimize the runtime overhead of building CCT through adaptive bursting. D'Elia et al. [19] optimize the space overhead by building Hot Calling Context Tree (HCCT) through mining frequent items in the calling context stream.

LagHunter [46] tries to provide lag information for interactive systems when they are handling user events. AppInsight [82] is built to monitor mobile applications that are asynchronized and multi-threaded. AppInsight can automatically instrument mobile binary codes and figure out critical path in each user-transaction across asynchronous-call boundaries. Changing the critical path will change the user-perceived latency. Leveraging critical-path information, developers can identify code regions to optimize and improve user experience. After evaluating four commonly used Java profilers, Mytkowicz et al. [70] find that these profilers often do not agree with each other, and do not always produce actionable profiles. There are two reasons causing this problem: first, the four profilers take samples only at yield points, and this is not randomly taking samples from program execution; second, compiler optimizations are affected by these profilers, and program performance and the placement of yield points are changed under profiling. The authors design and implement a new profiler, which does not suffer from the two problems.

Unlike our work, profiling aims to tell where computation resources are spent, not where and why computation resources are wasted. The root-cause code region of a performance problem often is not inside the top-ranked functions in the profiling result (Chapter 5). Even if it is, developers still need to spend a lot of effort to understand whether and what type of computation inefficiency exists.

2.3.2 Automated performance diagnosis tools

There are also tools proposed to diagnose certain types of performance problems.

X-ray [6] aims to diagnose performance problems caused by end users. The root causes discussed in the X-ray paper are unexpected inputs or configurations that can be changed by end users. X-ray pinpoints the inputs or configuration entries that are most responsible for performance problems and helps users to solve performance issues themselves (by changing the inputs or configuration entries). The main technique used in X-ray is called performance summarization, which first attributes a performance cost to each basic block, and then estimates the possibility that each block will be executed due to

certain input entry, and finally ranks all input entries. The diagnosis techniques discussed in this thesis aim to help developers. We want to provide information to help developers change inefficient codes and fix performance bugs.

Some diagnosis techniques are built through analyzing system log. IntroPerf [50] automatically infers latency of calling contexts, composed of user-level and kernel-level function calls, based on operating system tracers. After compared with normal runs, calling contexts from buggy runs will be ranked to help diagnose performance problems. StackMine [35] mines costly-maximal-pattern from stack traces collected from event handlers, whose response time is longer than a predefined threshold. Identified patterns will be clustered to help developers effectively analyze impactful performance bugs. A two-step technique proposed by Yu et al. [106] analyzes system traces to understand performance impact propagation and the causality relationship among different system components. In the impact analysis step, performance measurement is conducted for each component. In the causality analysis step, wait graphs are built for both slow and fast executions, and contrast data mining is used to identify special path segments on the slow wait graph. Too much logging would incur a large runtime overhead and cause an extra burden for log analysis. Log2 [20] tries to control the overhead of logging, while keeping its performance diagnosis capability. Log2 achieves this goal by making a "whether to log" decision for each log request through a two-phase filtering mechanism. During the local filter, log requests, whose utility scores lower than a global threshold, are discarded. In the global filter, all log requests are ranked, and only top-ranked requests are flushed to disk to meet the logging budget request. All these diagnosis tools are very useful in practice, but they have different focus from the diagnosis work in this thesis. They do not aim to identify source-code fine-grained root causes of performance problems reported by end users.

Many techniques have been proposed to diagnose performance problems in distributed systems [26, 48, 85, 105]. These techniques often focus on identifying the faulty components/nodes or faulty interactions that lead to performance problems, which are different from the diagnosis work in this thesis.

2.4 Other Techniques to Fight Performance Bugs

There are many test input generation techniques to improve performance testing process. Wise [10] learns how to restrict conditional branches from small inputs, and uses the learned policy to generate larger inputs with worst-case complexity. EventBreak [81] generates test inputs for web applications. SpeedGun [80] generates tests for changed concurrent classes.

Some techniques aim to improve the test selection or prioritize test targets during performance testing. Forepost [32] is a test-selection framework toward exposing more performance bottlenecks. It run applications under a small set of randomly selected test inputs to learn rules about which types of inputs are more likely to trigger intensive computation. Then it uses learned rules to pick remaining test inputs in order to expose performance bottlenecks quickly. Huang et al. [39] study 100 performance regression issues. Based on the studying results, they design a performance risk analysis, which can help developers prioritize performance testing targets.

Big data systems are mainly written in managed languages. They suffer from two performance issues [9, 71]: the excessive use of pointers or object reference will lead to a high space overhead, and frequent garbage collection (GC) will prevent the systems from making progress. To address these problems, Bu et al. [9] propose a design paradigm, which includes two components: merging small data record objects into a large buffer, and conducting data manipulation directly on the buffer. In a follow-up work, Facade [71], a compiler solution, is proposed to separate data store from data manipulation. After the transformation enforced by Facade, data is stored in off-heap native memory, and heap objects are only created for control purposes.

All these techniques combat performance bugs in different aspects from this thesis.

Chapter 3

Real-World Performance-Bug Understanding

Empirical studies on functional bugs have successfully guided the design of functional software testing, bug detection, and failure diagnosis. Poor understanding of performance bugs is part of the causes of today's performance-bug problems. The lack of empirical studies on performance bugs has severely limited the design of performance-bug avoidance, testing, detection, and fixing tools.

This chapter presents our comprehensive study of real-world performance bugs, based on 110 bugs randomly collected from the bug databases of five representative open-source software suites (Apache, Chrome, GCC, Mozilla, and MySQL). Following the lifetime of performance bugs, our empirical study is performed in 4 dimensions: the root-cause of performance bugs, how performance bugs are introduced, how to expose performance bugs, and how to fix performance bugs. Our findings and implications can guide future research in this area, and have already inspired our own performance-bug detection and performance failure diagnosis work.

3.1 Introduction

Performance bugs [13, 49, 67, 89] are software implementation mistakes that can cause slow execution. The patches for performance bugs are often not too complex. These patches can preserve the same functionality and achieve significant performance improvement. Performance bugs cannot be optimized away by compiler optimization. Many of them escape from testing process and manifest in front of end users [97]. Performance bugs are common and

severe. Mozilla developers have to fix 5–60 performance bugs each month in the past 10 years. Performance bugs have already caused several highly publicized failures, such as the slow affordable care act system [47]. Fighting performance bugs is solely needed.

There are many misunderstanding of performance bugs, such as "performance is taken care of by compilers and hardware" and "profiling is sufficient to solve performance problems". These wrong perceptions are partly the causes of today's performance-bug problem [23]. The lack of understanding of performance bugs has severely limited the research and tool building in this area.

There are many empirical studies on functional bugs [15, 62, 63, 78, 84, 92]. Many of them are conducted along the following dimensions: root causes of functional bugs, how functional bugs are introduced, how to expose functional bugs, and how to fix functional bugs. These studies have provided guidance for functional-bug detection, functional failure diagnosis, functional-bug avoidance, and software testing. It is feasible and necessary to conduct similar studies on performance bugs for the following reasons:

Firstly, it is feasible to sample performance-bug reports in the real world to conduct the empirical study. There are well-known open-source software suites with long well maintained bug databases. For some of them, developers explicitly mark certain bug reports in their bug databases as performance bugs. For example, Mozilla developers use "perf" tag to mark performance bugs. It is fairly easy for us to collect enough performance bugs to conduct the study.

Secondly, it is reasonable to understand whether techniques designed for functional bugs still work for performance bugs, before designing new techniques for performance bugs. In order to make this clear, it is necessary to follow study dimensions conducted on functional bugs to perform a similar empirical study for performance bugs.

Finally, performance bugs are different from functional bugs. For example, performance bugs do not have failure symptoms, such as crash and segmentation fault. We cannot directly leverage experience gained from combating functional bugs. It is necessary to understand the unique features and bug patterns for performance bugs.

This chapter makes the first, to the best of our knowledge, comprehensive study of real-world performance bugs based on 110 bugs randomly collected from the bug databases of five representative open-source software suites (Apache, Chrome, GCC, Mozilla, and MySQL). Our study has made the following findings.

Guidance for bug avoidance. Two-thirds of the studied bugs are introduced by developers' wrong understanding of workload or APIs' performance features. More than one quarter of the bugs arise from previously correct code due to workload or API changes. To avoid performance bugs, developers need performance-oriented annotation systems and change-impact analysis.

Guidance for performance testing. Almost half of the studied bugs require inputs with **both** special features and large scales to manifest. New performance-testing schemes that combine the input-generation techniques used by functional testing [11, 31] with a consideration towards large scales will significantly improve the state-of-the-art.

Guidance for bug detection. Recent works [13, 22, 49, 87, 100, 103] have demonstrated the potential of performance-bug detection. Our study found common root causes and structural patterns of real-world performance bugs that can help improve the coverage and accuracy of performance-bug detection.

Guidance for bug diagnosis. The root-cause patterns and fix strategies for performance bugs are highly correlated. It is feasible to propose fix strategies automatically, based on identified root causes.

Comparison with functional bugs. Performance bugs tend to hide for much longer time in software than functional bugs. Unlike functional bugs, performance bugs cannot all be modeled as rare events, because a non-negligible portion of them can be triggered by almost all inputs.

General motivation (1) Many performance-bug patches are small. The fact that we can achieve significant performance improvement through a few lines of code change motivates researchers to pay more attention to performance bugs. (2) A non-negligible portion of performance bugs in multi-threaded software are related to synchronization. Developers need tool support to avoid over-synchronization traps.

3.2 Methodology

This section describes how we collect performance bugs from the real world.

Applications We chose five open-source software suites to examine: Apache, Chrome, GCC, Mozilla, and MySQL. These popular, award-winning software suites [40] are all large-scale and mature, with millions of lines of source code and well maintained bug databases.

Application Suite Description (language)	# of Bugs
Apache Suite HTTPD: Web Server (C) TomCat: Web Application Server (Java) Ant: Build management utility (Java)	25
Chromium Suite Google Chrome browser (C/C++)	10
GCC Suite GCC & G++ Compiler (C/C++)	11
Mozilla Suite Firefox: Web Browser (C++, JavaScript) Thunderbird: Email Client (C++, JavaScript)	36
MySQL Suite Server: Database Server (C/C++) Connector: DB Client Libraries (C/C++/Java/.Net)	28
Total	110

Table 3.1: Applications and bugs used in the study.

As shown in Table 3.1, these five suites provide a good coverage of various types of software, such as interactive GUI applications, server software, command-line utilities, compilers, and libraries. They are primarily written in C/C++ and Java. Although they are all open-source software, Chrome is backed up by Google and MySQL was acquired by Sun/Oracle in 2008. Furthermore, the Chrome browser was first released in 2008, while the other four have had 10–15 years of bug reporting history. From these applications, we can observe both traditions and new software trends such as web applications.

Bug Collection GCC, Mozilla, and MySQL developers *explicitly* mark certain reports in their bug databases as performance bugs using special tags, which are *compile-time-hog*, *perf*, and *S5* respectively. Apache and Chrome developers do not use any special tag to mark performance bugs. Therefore,

we searched their bug databases using a set of performance-related keywords ('slow', 'performance', 'latency', 'throughput', etc.).

From these sources, we *randomly* sampled 110 fixed bugs that have sufficient documentation. Among these bugs, 44 were reported after 2008, 39 were reported between 2004 and 2007, and 27 were reported before 2004. 41 bugs came from server applications and 69 bugs came from client applications. The details are shown in Table 3.1.

Caveats Our findings need to be taken with the methodology in mind. The applications in our study cover representative and important software categories, workload, development background, and programming languages. Of course, there are still uncovered categories, such as scientific computing software and distributed systems.

The bugs in our study are collected from five bug databases without bias. We have followed the decisions made by developers about what are performance bugs, and have not intentionally ignored any aspect of performance problems in bug databases. Of course, some performance problems may never be reported to the bug databases and some reported problems may never be fixed by developers. Unfortunately, there is no conceivable way to study these unreported or unfixed performance problems. We believe the bugs in our study provide a representative sample of the reported and fixed performance bugs in these representative applications.

We have spent more than one year to study all sources of information related to each bug, including forum discussions, patches, source code repositories, and others. Each bug is studied by at least two people and the whole process consists of several rounds of bug (re-)study, bug (re-)categorization, cross checking, etc.

Finally, we do not emphasize any quantitative characteristic results, and most of the characteristics we found are consistent across all examined applications.

3.3 Case Studies

We will discuss six motivating examples in this part, and we will answer the following questions by using these six examples.

```
//Apache#45464
//modules/dav/fs/repos.c

status = apr_stat(&fsctx->info1.finfo, fsctx->path1.buf,
-                 APR_FINFO_NORM | APR_FINFO_LINK, pool);
+                 APR_FINFO_TYPE | APR_FINFO_LINK, pool);
```

Figure 3.1: An Apache-HTTPD bug retrieving more than necessary data

```
//Mozilla#66461 & Patch
//gfx/src/gtk/nsImageGTK.cpp

//When the input figure is transparent, mIsSpacer will be true.
//The patch conditionally skips nsImageGTK::Draw().
NS_IMETHODIMP nsImageGTK::Draw(...)
{
  ...
+ if ((mAlphaDepth==1) && mIsSpacer)
+   return NS_OK;
  ...
  //render the input figure
}
```

Figure 3.2: A Mozilla bug drawing transparent figures

(1) Are performance bugs too different from traditional bugs to study along the traditional bug-fighting process (i.e., bug avoidance, testing, detection, and fixing)?

(2) If they are not too different, are they too similar to be worthy of a study?

(3) If developers were more careful, do we still need research and tool support to combat performance bugs?

Retrieve Unnecessary (Figure 3.1) Apache HTTPD developers forgot to change a parameter of API apr_stat after an API upgrade. This mistake causes apr_stat to retrieve more than necessary information from the file system and leads to more than ten times slowdown in Apache server. After changing the parameter, apr_stat will retrieve exactly what developers originally needed

Transparent Draw (Figure 3.2) Mozilla developers implemented a procedure nsImageGTK::Draw for figure scaling, compositing, and rendering, which is a waste of time for transparent figures. This problem did not catch de-

```
//Mozilla515287 & Patch
//content/base/src/nsXMLHttpRequest.cpp

//This was not a bug until Web 2.0, where doing
//garbage collection (GC) after every XMLHttpRequest(XHR)
//is too frequent.
nsXMLHttpRequest::OnStopRequest(...)
{
  ...
-  mScriptContext->GC();
}
```

Figure 3.3: A Mozilla bug doing intensive GCs

velopers' attention until two years later when 1 pixel by 1 pixel transparent GIFs became general purpose spacers widely used by Web developers to work around certain idiosyncrasies in HTML 4. The patch of this bug skips nsImageGTK::Draw when the function input is a transparent figure.

Intensive GC (Figure 3.3) Users reported that Firefox cost 10 times more CPU than Safari on some popular Web pages, such as gmail.com. Lengthy profiling and code investigation revealed that Firefox conducted an expensive garbage collection (GC) process at the end of *every* XMLHttpRequest, which is too frequent. A developer then recalled that GC was added there five years ago when XHRs were infrequent, and each XHR replaced substantial portions of the DOM in JavaScript. However, things have changed in modern Web pages. As a primary feature enabling Web 2.0, XHRs are much more common than they were five years ago. This bug is fixed by removing the call to GC.

Bookmark All (Figure 3.4) Users reported that Firefox hung when they clicked 'bookmark all (tabs)' with 50 open tabs. Investigation revealed that Firefox used N database transactions to bookmark N tabs, which is very time-consuming comparing with batching all bookmark tasks into a single transaction. Discussion among developers revealed that there was almost no batchable database task in Firefox a few years back. The addition of batchable functionalities, such as 'bookmark all (tabs)' exposed this inefficiency problem. After replacing N invocations of doTransact with a single doAggregateTransact, the

```
//Mozilla#490742 & Patch
//browser/components/places/src/nsPlacesTransactionsService.js

//doTransact saves one tab into 'bookmark' SQLite Database.
//Firefox hangs @ 'bookmark all (tabs)'
for (var i = 0; i < tabs.length; ++i)
{
  ...
-  tabs[i].doTransact();
}

+ doAggregateTransact(tabs);
```

Figure 3.4: A Mozilla bug with un-batched DB operations

```
//MySQL#38941 & Patch
//mysys/thr_mutex.c

//random() is a serialized global−mutex−protected
//glibc function.
int fastmutex_lock(...)
{
  ...
-  maxdelay += (double)random();
+  maxdelay += (double)park_rng();
  ...
}
```

Figure 3.5: A MySQL bug with over synchronization

hang disappears. During patch review, developers found two more places with similar problems and fixed them by doAggregateTransact.

Slow Fast-Lock (Figure 3.5) In order to conduct fast locking, MySQL synchronization-library developers implemented a fastmutex_lock, which would call library function random to calculate spin delay. Unfortunately, it turns out that random actually contains a lock, and this lock serializes every thread that invoke random. Developers' unit test showed that invoking random from multi-thread could be 40 times slower than from multi-process, due to the lock contention. This bug is fixed by replacing random with a non-synchronized random number generator.

```
//MySQL#26527 & Patch
//sql/ha_partition.cc

void ha_partition::start_bulk_insert(ha_rows rows)
{
   ...
-    if (!rows)
-    {   //slow path where caching is not used
-       DBUG_VOID_RETURN;
-    }
-    rows = rows/m_tot_parts + 1;
+    rows = rows ? (rows/m_tot_parts + 1) : 0;
   ...
   //fast path where caching is used
   DBUG_VOID_RETURN;
}
```

Figure 3.6: A MySQL bug without using cache

No Cache (Figure 3.6) MySQL users reported that loading data into a partitioned table would be 20 times slower, compared with loading the same amount of data into an unpartitioned table. The slowness comes from the fact that cache was not used, and it is the branch in Figure 3.6 causing cache not to be allocated. The developer who implemented this start_bulk_insert function thought that parameter 0 indicates no need of cache, while developer who wrote caller function thought that parameter 0 means the allocation of a large cache. This miscommunication causes this bug. The patch is to change the branch selection when using 0 as parameter.

These six bugs can help us answer the questions asked earlier.

(1) They have similarity with traditional bugs. For example, they are either related to usage rules of functions/APIs or related to programs' control flow, like branch, both of which are well studied by previous work on detecting and diagnosing functional bugs [55, 57, 61].

(2) They also have interesting differences compared to traditional bugs. For example, the code snippets in Figure 3.1–3.4 turned buggy (or buggier) long after they were written, which is rare for functional bugs. As another example, testing designed for functional bugs cannot effectively expose bugs like *Bookmark All*. Once the program has tried the 'bookmark all (tab)' button

Root Causes for Performance Bugs	Apache	Chrome	GCC	Mozilla	MySQL	Total
Wrong Branch: branch selection leading to performance loss	2	0	2	7	6	17
Resultless: not generating desired results	3	5	4	20	10	42
Redundancy: generating the same results repeatedly	13	3	4	6	5	31
Synchronization Issues: inefficient synchronization among threads	6	1	0	1	8	16
Others: all the bugs not belonging to the above four categories	5	1	1	3	3	13

Table 3.2: Root cause categorization in Section 3.4.

with one or two open tabs, bookmarking more tabs will not improve the statement or branch coverage and will be skipped by functional testing.

(3) Developers cannot fight these bugs by themselves. They cannot predict future workload or code changes to avoid bugs like *Retrieve Unnecessary*, *Transparent Draw*, *Intensive GC*, and *Bookmark All*. Even experts who implemented synchronization libraries could not avoid bugs like *Slow Fast-Lock*, given opaque APIs with unexpected performance features. Research and tool support are needed here.

Of course, it is premature to draw any conclusion based on six bugs. Next, we will comprehensively study 110 performance bugs.

3.4 Root Causes of Performance Bugs

There are a large variety of potential root causes for inefficient code, such as poorly designed algorithms, non-optimal data structures, cache-unfriendly data layouts, etc. Our goal here is not to discover previously unheard-of root causes, but to check whether there are common root-cause patterns among real-world performance bugs that bug detection and diagnosis work can focus on.

Our study shows that the majority of real-world performance bugs in our study are covered by only a couple of root-cause categories (Table 3.2).

Wrong Branch A non-negligible portion of performance bugs are branch-related. There are three situations for bugs under this category. Firstly, wrong branches lead to some slow code paths. For example, when Mozilla#231300 is triggered, Firefox would use separate system calls to move files in the same directory one by one instead of using one single system call to move them

altogether. Seconly, wrong branches lead to unnecessary computation or execution. For example, when Mozilla#258793 is triggered, Firefox will call draw functions for background figures, which actually do not exist. Finally, wrong branches lead to inefficient functionalities, such as the *No Cache* example shown in Figure 3.6.

Resultless Around one-third of performance bugs are caused by resultless codes. Buggy codes rarely generate results when these bugs are triggered. Bugs in this category can be further categorized along two dimensions: according to different granularities, resultless bugs can be divided into loop-related bugs and not loop-related bugs; based on whether semantic information is needed to identify resultless, resultless bugs can be divided into semantic resultless, and non-semantic resultless. When *Intensive GC* shown in Figure 3.3 triggered, the loop conducting garbage collection scans all heap objects, but rarely finds free objects with reference number 0 and deallocate them. This bug is loop-related and non-semantic. *Transparent Draw* in Figure 3.2 is not loop-related, and semantic information is needed to know that drawing a transparent figure does not generate any results.

Redundancy Redundancy means generating the same results repeatedly. Intuitively, we can remove repeated work and improve performance. According to different code granularity to observe redundant work, bugs in this category can be divided into redundant snippets, cross-iteration redundancy, and cross-loop redundancy. For example, in Chrome#70153, both software and GPU will render the same video redundantly, and this bug is categorized as redundant snippets. In the *Bookmark All* example shown in Figure 3.3, Firefox will start, commit, and destroy a transaction for each tab in each iteration, and there is a lot of redundant work across different iterations. There are also bugs caused by cross-loop redundancy, like Mozilla#35294

Synchronization Issues Unnecessary synchronization that intensifies thread competition is also a common root cause, as shown in the *Slow Fast-Lock* bug (Figure 3.6). These bugs are especially common in server applications, contributing to 6 out of 16 Apache server bugs and 7 out of 25 MySQL server bugs.

How Performance Bugs Are Introduced	Apache	Chrome	GCC	Mozilla	MySQL	Total
Workload Issues: developers' workload assumption is wrong or out-dated	15	4	7	21	10	57
API Issues: misunderstand performance features of functions/APIs	6	2	1	10	9	28
Others: all the bugs not belonging to the above two categories	4	4	3	6	9	26

Table 3.3: How performance bugs are introduced in Sections 3.5.

Others There are also bugs that do not fall into above categories. For these bugs, developers find more efficient methods to optimize original codes. For example, in order to accelerate the slow startup of GPU process reported in Chrome#59711, developers use one extra thread to collect expensive GPU information. For MySQL#14637, MySQL developers replace byte-wise parsing by using four-byte-wise parsing to accelerate trimming blank characters from the end of a string.

3.5 How Performance Bugs Are Introduced

We have studied the discussion among developers in bug databases and checked the source code of different software versions to understand how bugs are introduced. Our study has particularly focused on the challenges faced by developers in writing efficient software, and features of modern software that affect the introduction of performance bugs.

Our study shows that developers are in a great need of tools that can help them avoid the following mistakes.

Workload Mismatch Performance bugs are most frequently introduced when developers' workload understanding does not match with the reality.

Our further investigation shows that the following challenges are responsible for most workload mismatches.

Firstly, the input paradigm could shift *after* code implementation. For example, the HTML standard change and new trends in web-page content led to *Transparent Draw* and *Intensive GC*, shown in Figure 3.2 and Figure 3.3.

Secondly, software workload has become much more diverse and complex than before. A single program, such as Mozilla, may face various types of workload issues: the popularity of transparent figures on web pages led to *Transparent Draw* in Figure 3.2; the high frequency of XMLHttpRequest led to

Intensive GC in Figure 3.3; users' habit of not changing the default configuration setting led to Mozilla#110555.

The increasingly dynamic and diverse workload of modern software will lead to more performance bugs in the future.

API Misunderstanding The second most common reason is that developers misunderstand the performance feature of certain functions. This occurs for 28 bugs in our study.

Sometimes, the performance of a function is sensitive to the value of a particular parameter, and developers happen to use performance-hurting values.

Sometimes, developers use a function to perform task i, and are unaware of an irrelevant task j conducted by this function that hurts performance but not functionality. For example, MySQL developers did not know the synchronization inside random and introduced the *Slow Fast-Lock* bug shown in Figure 3.5.

Code encapsulation in modern software leads to many APIs with poorly documented performance features. We have seen developers explicitly complain about this issue [91]. It will lead to more performance bugs in the future.

When a bug was not buggy An interesting trend is that 29 out of 110 bugs were not born buggy. They became inefficient long after they were written due to workload shift, such as that in *Transparent Draw* and *Intensive GC* (Figures 3.2 and 3.3), and code changes in other part of the software, such as that in Figure 3.1. In Chrome#70153, when GPU accelerator became available, software rendering code became redundant. Many of these bugs went through regression testing without being caught.

3.6 How Performance Bugs Are Exposed

We define exposing a performance bug as causing a perceivably negative performance impact, following the convention used in most bug reports. Our study demonstrates several unique challenges for performance testing.

Always Active Bugs A non-negligible portion of performance bugs are almost always active. They are located at the start-up phase, the shutdown phase, or other places that are exercised by almost all inputs. They could

How Performance Bugs Are Exposed	Apache	Chrome	GCC	Mozilla	MySQL	Total
Always Active: almost every input on every platform can trigger this bug	2	3	0	6	5	16
Special Feature: need special-value inputs to cover specific code regions	18	7	11	23	17	76
Special Scale: need large-scale inputs to execute a code region many times	18	2	10	21	18	69
Feature+Scale: the intersection of Special Feature and Special Scale	13	2	10	14	12	51

Table 3.4: How performance bugs are exposed in Section 3.6.

be very harmful in the long term, because they waste performance at every deployment site during every run of a program. Many of these bugs were caught during comparison with other software (e.g., Chrome vs. Mozilla vs. Safari).

Judging whether performance bugs have manifested is a unique challenge in performance testing.

Input Feature & Scale Conditions About two-thirds of performance bugs need inputs with special features to manifest. Otherwise, the buggy code units cannot be touched. Unfortunately, this is not what black-box testing is good at. Much manual effort will be needed to design test inputs, a problem well studied by past research in functional testing [11, 13].

About two-thirds of performance bugs need large-scale inputs to manifest in a perceivable way. These bugs cannot be effectively exposed if software testing executes each buggy code unit only once, which unfortunately is the goal of most functional testing.

Almost half of the bugs need inputs that have special features **and** large scales to manifest. For example, to trigger the bug shown in Figure 3.3, the user has to click 'bookmark all' button (i.e., special feature) with many open tabs (i.e., large scale).

3.7 How Performance Bugs Are Fixed

There are four common strategies in fixing performance bugs, as shown in Table 3.5.

Change Condition The most common fix strategy is *Change Condition*. It is used in 36 patches, in which code units not always generating results are conditionally skipped, a fast path is changed to be executed, or the same func-

How to Fix Performance Bugs	Apache	Chrome	GCC	Mozilla	MySQL	Total
Change Condition: a condition is added or modified	3	3	5	13	9	36
In-place Call Change: replace call sequences in the exact same place	5	1	0	12	10	28
Memoization: reuse results from previous computation	9	1	3	2	3	18
Batch: batch computation to eliminate redundancy	4	3	1	6	1	15
Others: all the bugs not belonging to the above four categories	4	2	2	3	5	16

Table 3.5: How to fix performance bugs in Section 3.7.

tionality is realized in a more efficient way. For example, Draw is conditionally skipped to fix *Transparent Draw* (Figure 3.2), and cache will be used to fix *No Cache* (Figure 3.6).

In-place Call Change The second most common strategy is *In-place Call Change*. By using this strategy, developers replace or reorganize the call sequence in the exact same place. The performance gain can be achieved whenever changed codes are executed. For example, in order to fix Mozilla#103330, developers replace the setLength and Append call combination with Assign.

Memoization fixes 18 bugs by reusing results from previous computation. For example, in order to fix Mozilla#409961, developers move do_QueryInterface outside the buggy loop, instead of calling it in each iteration.

Batch strategy is used in 15 patches. For example, *Bookmark All* in Figure 3.4 is fixed by this strategy.

Are patches complicated? Most performance bugs in our study can be fixed through simple changes. In fact, 41 out of 110 bug patches contain five or fewer lines of code changes. The median patch size for all examined bugs is 9 lines of code. The small patch size is a result of the above fixing strategies. Many change condition patches are small. Most in-place call change, memoization, and batch patches do not require implementing new function calls.

3.8 Other Characteristics

Lifetime We chose Mozilla to investigate the lifetime of performance bugs, due to its convenient CVS query interface. We consider a bug's life to have started when its buggy code was first written. The 36 Mozilla bugs in our study took 966 days on average to get discovered, and another 140 days on

average to be fixed. For comparison, we randomly sampled 36 functional bugs from Mozilla. These bugs took 252 days on average to be discovered, which is much shorter than that of performance bugs in Mozilla. These bugs took another 117 days on average to be fixed, which is a similar amount of time with those performance bugs.

Location For each bug, we studied the location of its minimum unit of inefficiency. Our first finding shows that most performance bugs happen at call sites, and their fix are changing the usage of function calls, such as replacing old call sequences with new call sequences, conditionally or unconditionally skipping buggy functions or changing parameters, and so on. For example, *Retrieve Unnecessary* (Figure 3.1), *Transparent Draw* (Figure 3.2), *Intensive GC* (Figure 3.3), *Bookmark All* (Figure 3.4), and *Slow Fast-Lock* (Figure 3.5) are all fixed by changing function-call usage. This is probably because developers and compilers have already done a good job in optimizing code within each procedure. Therefore, future work to detect, diagnose and fix performance bugs should allocate more effort at call sites and procedure boundaries.

There are also 32 bugs not fixed by changing function-call usage. These bugs mainly arise from two scenarios. In one scenario, the buggy code unit itself does not directly waste performance. Instead, its impact propagates to other places in the software and causes performance loss there. For example, the *No Cache* (Figure 3.6) bug happens when MySQL mistakenly does not allocate cache. This operation itself does not take time, but it causes performance loss later. The second scenario is to optimize code units inside functions, like MySQL#14637, whose patch replaces byte-wise parsing with four-byte-wise parsing to accelerate trimming blank characters from the end of a string.

Our second finding shows that around three quarters of bugs are located inside either an input-dependent loop or an input-event handler. For example, the buggy code in Figure 3.3 is executed at every XHR completion. The bug in Figure 3.2 wastes performance for every transparent image on a web page. In addition, about half performance bugs involve I/Os or other time-consuming system calls. There are a few bugs whose buggy code units only execute once or twice during each program execution. For example, the Mozilla#110555 bug wastes performance while processing exactly two fixed-size default con-

figuration files, userChrome.css and userContent.css, during the startup of a browser.

Correlation Among Categories Following previous empirical studies [54], we use a statistical metric *lift* to study the correlation among characteristic categories. The *lift* of category A and category B, denoted as *lift(AB)*, is calculated as $\frac{P(AB)}{P(A)P(B)}$, where P(AB) is the probability of a bug belonging to both categories A and B. When *lift(AB)* equals 1, category A and category B are independent of each other. When *lift(AB)* is greater than 1, categories A and B are positively correlated: when a bug belongs to A, it likely also belongs to B. The larger the *lift* is, the more positively A and B are correlated. When *lift(AB)* is smaller than 1, A and B are negatively correlated: when a bug belongs to A, it likely does not belong to B. The smaller the *lift* is, the more negatively A and B are correlated.

Root cause categories are highly correlated with fix strategies. Among all correlations, the redundant root cause and the memoization fix strategy are the most positively correlated with a 3.54. The wrong branch selection root cause is strongly correlated with the change condition fix strategy with a 2.74 lift. The redundant root cause and the batch fix strategy are the third most positively correlated pair with a 2.36 lift. On the other hand, the wrong branch selection root cause has the most negative correlation with in-place call change, memoization and batch bug-fix strategies. Their lifts are all 0.

Server Bugs vs. Client Bugs Our study includes 41 bugs from server applications and 69 bugs from client applications. To understand whether these two types of bugs have different characteristics, we apply chi-square test [95] to each category listed in Table 3.2, Table 3.3, Table 3.4 and Table 3.5. We choose 0.01 as the significance level of our chi-square test. Under this setting, if we conclude that server and client bugs have different probabilities of falling into a particular characteristic category, this conclusion only has a 1% probability to be wrong.

We find that, among all the categories listed in Table 3.2, Table 3.3, Table 3.4 and Table 3.5, only the synchronization issues category is significantly different between server bugs and client bugs — synchronization issues have caused 31.7% of server bugs and only 4.3% of client bugs.

3.9 Guidance for My Thesis Work

Performance-bug Detection Our study provides several motivations for our own rule-based bug detection work, which will be discussed in detail in Chapter 4. Most performance bugs loss performance at function-call sites (Section 3.8), more than one-fourth of performance bugs are introduced by misunderstanding API (Section 3.5), and more than one-fourth of performance bugs are fixed by in-place call changes (Section 3.7). We could detect similar inefficient call usage to find new bugs, and propose more efficient call sequences with the same functionalities. Because some performance bugs are always active (Section 3.6), performance bugs cannot be modeled as rare events. Automatically inferring efficiency rules [24] may not be feasible for performance bugs. Patches for performance bugs are simple (Section 3.7), and they follow limited fix strategies. It is feasible to extract efficiency rules from these patches.

Performance Failure Diagnosis Our study also provides guidance for our performance failure diagnosis work (Chapter 5 and Chapter 6). A non-negligible portion of performance bugs are caused by wrong branch selection (Section 3.5). Statistical debugging, leveraging branch predicate, can well diagnose functional bugs with similar root causes. It is promising to explore how to apply statistical debugging to diagnose performance bugs. The three common root causes are wrong branch, resultless and redundancy (Section 3.4). Our diagnosis projects should focus on bugs caused by these three root causes. Root causes and fix strategies are highly correlated (Section 3.8). It is feasible to automatically provide fix suggestions based on identified root causes.

3.10 Guidance for Future Work

Comparison with Functional Bugs There are several interesting comparisons between performance and functional bugs. (1) The distribution of performance-failure rates over software life time follows neither the bathtub model of hardware errors nor the gradually maturing model of functional bugs, because performance bugs have long hiding periods (Section 3.8) and can emerge from non-buggy places when software evolves (Section 3.5). (2) Unlike functional bugs, performance bugs cannot always be modeled as rare events, because

some of them are always active (Section 3.6). (3) The percentage of synchronization problems among performance bugs in our study is higher than the percentage of synchronization problems among functional bugs in a previous study for a similar set of applications [54] (Section 3.4).

Annotation Systems Annotation systems are used in many software development environments [66, 93]. Unfortunately, they mainly communicate functionality information.

Our study calls for performance-aware annotation systems [79, 94] that help developers maintain and communicate APIs' performance features and workload assumptions (Section 3.5). Simple support such as warning about the existence of locks in a library function, specifying the complexity of a function, and indicating the desired range of a performance-sensitive parameter can go a long way in avoiding performance bugs. Recent work that automatically calculates function complexity is also promising [33].

Testing Regression testing and change-impact analysis have to consider workload changes and performance impacts, because new performance bugs may emerge from old code (Section 3.5).

Performance testing can be improved if its input design combines smart input-generation techniques used in functional testing [11, 31] with an emphasis on large scale (Section 3.6).

Expressing performance oracles and judging whether performance bugs have occurred are critical challenges in performance testing (Section 3.6). Techniques that can smartly compare performance numbers across inputs and automatically discover the existence of performance problems are desired.

Future Directions One might argue that performance sometimes needs to be sacrificed for better productivity and functional correctness. However, the fact that we can often achieve significant performance improvement through only a few lines of code change motivates future research to pay more attention to performance bugs (Section 3.7). Our study suggests that the workload trend and API features of modern software will lead to more performance bugs in the future (Section 3.5). In addition, our study observes a significant portion of synchronization-related performance bugs in multi-threaded software. There will be more bugs of this type in the multi-core era. Beyond research discussed

in this proposal, we think that there are still several potential directions to combat performance bugs.

Finally, our observations have been consistent across old software and new software (Chrome), old bugs (27 pre-2004 bugs) and new bugs (44 post-2008 bugs). Therefore, we are confident that these lessons will be useful at least for the near future.

3.11 Conclusions

Performance bugs have largely been ignored in previous research on software defects. Facing the increasing significance of performance bugs, this chapter provides one of the first studies on real-world performance bugs based on 110 bugs collected from five representative software suites. The study covers a wide spectrum of characteristics, and provides guidance for future research on performance-bug avoidance, performance testing, bug detection, etc. However, the empirical studies presented in this chapter do not cover all characteristics of real-world performance bugs that might be interesting for tool developers. In fact, later in Section 5.2 and Section 6.2, we will further study subsets of these performance bugs to guide our research in performance failure diagnosis.

Chapter 4

Rule-Based Performance-Bug Detection

Bug detection aims to find previously unknown bugs inside programs. This chapter presents our rule-based static performance-bug detection work. Guided by the characteristics study in the last chapter, we extract efficiency rules from 25 patches and use them to detect new performance bugs. 332 previously unknown performance problems are found in the latest versions of Apache, Mozilla, and MySQL applications, including 219 performance problems found by applying rules across applications.

4.1 Introduction

Rule-based detection approach is effective for discovering functional bugs and security vulnerabilities [14, 27, 38, 55, 69]. Many functional bugs can be identified by comparing against certain function-call sequences that have to be followed in a program for functional correctness and security.

We hypothesize that rule-based bug detection is useful for detecting performance bugs based on our characteristics study:

Efficiency rules should exist. Many performance bugs happen at call sites (Section 3.8), and those inefficient call sites could all become rules. For example, `random()` should not be used by concurrent threads, and `doTransact()` in loops should be replaced by `aggregateTransact()`.

Efficiency rules can be easily collected from patches, as most patches are small and follow regular fixing strategies (Section 3.7). It will not be difficult to examine them and extract efficiency rules.

Efficiency rules could be widely applicable, as a misunderstanding of an API or workload could affect many places and lead to many bugs, considering how bugs are introduced (Section 3.5).

To test these hypotheses, we collected 25 rules from Apache, Mozilla, and MySQL bug patches, and built static checkers to detect rule violations. Our checkers automatically found 125 *potential performance problems* (*PPP*s) in the original buggy versions of Apache, Mozilla, and MySQL. Programmers failed to fix them together with the original 25 bugs where the rules came from. Our checkers also found 332 previously unknown PPPs in the latest versions of Apache, Mozilla, and MySQL, including 219 PPPs found by extracting rules from one application and applying rules to a *different* application. Our thorough code reviews and unit testings demonstrate that each PPP runs significantly slower than its functionality-preserving alternate suggested by the checker. 77 of found PPPs are already confirmed by developers and 15 of found PPPs are fixed based on our report.

The main contribution of our bug-detection work is that it confirms the existence and value of efficiency rules: efficiency rules in our study are usually violated at more than one place, by more than one developer, and sometimes in more than one program. Our experience motivates future work to automatically generate efficiency rules, through new patch languages [78], automated patch analysis [65], source code analysis, or performance-oriented annotations. Future work can also improve the accuracy of performance-bug detection by combining static checking with dynamic analysis and workload monitoring.

4.2 Efficiency Rules in Patches

Terminology *Efficiency rules*, or *rules*, include two components: a *transformation* and a *condition* for applying the transformation. Once a code region satisfies the condition, the transformation can be applied to improve performance and preserve functionality.

We have manually checked all the 110 performance-bug patches. 50 out of these 110 patches contain efficiency rules. The other 60 do not contain rules, because they either target too specific program contexts or are too general to be useful for rule-based bug detection.

Call Sequence Conditions
function f1 is invoked
function f1 is always followed by f2
function f1 is called once in each iteration of a loop
Parameter/Return Conditions
nth parameter of f1 equals K (constant)
nth parameter of f1 is the same variable as the return of f2
a param. of f1 and a param. of f2 point to the same object
the return of f1 is not used later
the parameter of f1 is not modified within certain scope
the input is a long string
Calling Context Conditions
function f1 is only called by one thread
function f1 can be called simultaneously by multiple threads
function f1 is called many times during the execution

Table 4.1: Typical conditions in function rules.

The conditions for applying these rules are composed of conditions on function-call sequences, parameter/return variables, and calling contexts, as shown in Table 4.1. For example, to apply the *Bookmark All* patch in Figure 3.4 elsewhere, one needs to find places that call `doTransact` inside a loop; to apply the patch in Figure 3.1 elsewhere, one needs to ensure that certain fields of the object pointed by the first parameter of `apr_stat` is not used afterward. There are also non-function rules, usually containing *Change Condition* transformation and other miscellaneous algorithm improvements.

4.3 Building Rule Checkers

Selecting Statically Checkable Rules Some rules' applying conditions are statically checkable, such as function f1 inside a loop; some are dynamically checkable, such as function f1 called by multiple threads at the same time; some are related to workload, such as having many large input files.

We check three largest application suites in our study: Apache, MySQL, and Mozilla. We find that 40 bug patches from them contain rules. 25 out of these 40 have applying conditions that are mostly statically checkable. Therefore, we have built checkers based on these **25** efficiency rules.

Checker Implementation We build 25 checkers in total. 14 of them are built using LLVM compiler infrastructure [52] for rules from C/C++ appli-

cations. LLVM works well for C++ software that troubles many other static analysis infrastructure [78]. It also provides sufficient data type, data flow, and control flow analysis support for our checking. The other 11 checkers are written in Python for 11 rules from Java, JavaScript, and C# applications.

The checker implementation is mostly straightforward. Each checker goes through software bitcode, in case of LLVM checkers, or source code, in case of Python checkers, looking for places that satisfy the patch-applying condition. We briefly discuss how our checkers examine typical conditions for function rules in the following.

Checking call-sequence conditions, exemplified in Table 4.1, involve mainly three tasks: (1) Differentiating functions with the same name but different classes; (2) Collecting loop information (loop-head, loop-exit conditions, loop-body boundaries, etc.); (3) Control flow analysis. LLVM provides sufficient support for all these tasks. Checkers written in Python struggle from time to time.

Checking parameter/return conditions, exemplified in Table 4.1, typically rely on data-flow analysis. In our current prototype, LLVM checkers conduct intra-procedural data-flow analysis. This analysis is scalable, but may lead to false positives and negatives. In practice, it works well as shown by our experimental results. Our current Python checkers can extract parameters of particular function calls, but can only do preliminary data-flow analysis.

4.4 Rule-Checking Methodology

We conduct all the experiments on an 8-core Intel Xeon machine running Linux version 2.6.18.

We apply every checker to the following software:

(1) The exact version of the software that the original patch was applied to, which is referred to as *original version*;

(2) The latest version of the software that the original patch was applied to, which is referred to as *original software*;

(3) The latest versions of software applications that are **different** from the one that the original patch was applied to, which is referred to as *different software*. This was applied to 13 checkers, whose rules are about *glibc* library

functions, Java library functions, and some general algorithm tricks. We will refer to this as *cross-application checking*. For example, a C/C++ checker from MySQL will be applied to Mozilla and Apache HTTPD for cross-application checking; a Java checker from Apache TomCat server will be applied to the 65 other Java applications in the Apache software suite[1].

The checking results are categorized into three types: *PPPs*, *bad practices*, and *false positives*. As discussed in Section 4.1, a PPP is an inefficient code region that runs slower than its functionality-preserving alternate implied by the efficiency rule. A bad practice is a region prone to becoming inefficient in the future. We reported some PPPs to developers. Based on our reports, 77 PPPs detected by 10 different checkers have been confirmed by the developers. Among confirmed PPPs, 15 PPPs detected by 7 different checkers have been fixed by the developers. Other reported PPPs are put on hold due to lack of bug-triggering input information, which is unfortunately out of the scope of this work.

Finally, we have also changed each checker slightly to report code regions that follow each efficiency rule. We refer to these regions as *good practices*, the opposite of PPPs.

4.5 Rule-Checking Results

Overall Results As shown in Table 4.2, 125 PPPs are found in the *original version* of software. Programmers missed them and failed to fix them together with the original bugs.

113 previously unknown PPPs are found in the latest versions of the *original software*, including bugs inherited from the original version and bugs newly introduced. Figure 4.1 shows an example.

219 previously unknown PPPs are found in the latest versions of *different software*. An example is shown in Figure 4.2.

77 PPPs in the latest versions of Apache, Mozilla, and MySQL are already confirmed by developers. 15 PPPs are already fixed by developers based on our reports.

[1]Development teams behind different Apache applications are different

ID	Orig. Buggy Version				Lastest Version of Same Softw.				Latest Version of Diff. Softw.				
	PPP	BadPr	F.P.	GoodPr	*PPP*	BadPr	F.P.	GoodPr	*PPP*	BadPr	F.P.	GoodPr	
Mozilla 35294	*5*	0	10	/	-	-	-	/	-	-	-	/	C++
Mozilla103330	*2*	0	0	117	*0*	0	0	7	-	-	-	-	C++
Mozilla258793	*1*	0	2	0	*0*	1	1	2	-	-	-	-	C++
Mozilla267506	*6*	0	0	9	*3*	0	0	19	-	-	-	-	C++
Mozilla311566	*26*	0	7	0	*25*	0	8	2	-	-	-	-	C++
Mozilla104962	*0*	0	0	1	*3*	0	0	12	*0*	0	0	0	C#
Mozilla124686	*0*	1	0	14	*0*	0	0	1	*0*	0	0	0	C#
Mozilla490742	*1*	0	3	5	*0*	0	0	4	-	-	-	-	JS
MySQL14637	*50*	0	11	/	*49*	0	11	/	*46*	0	31	/	C/C++
MySQL15811	*15*	20	5	5	*16*	20	7	7	-	-	-	-	C++
MySQL38769	*0*	0	1	5	-	-	-	-	-	-	-	-	C++
MySQL38941	*1*	4	0	2	*1*	4	0	2	*3*	5	2	0	C/C++
MySQL38968	*3*	0	1	38	*2*	0	2	43	-	-	-	-	C/C++
MySQL39268	*7*	0	0	4	*7*	0	0	18	-	-	-	-	C++
MySQL49491	*1*	0	0	0	*1*	0	0	2	*3*	0	0	0	C/C++
MySQL26152	*0*	0	0	0	*0*	0	0	0	*0*	0	1	4	C#
MySQL45699	*0*	2	0	0	*0*	0	0	0	*9*	0	0	45	C#/Java
Apache33605	*0*	2	0	/	*0*	2	0	/	*0*	5	0	/	C
Apache45464	*3*	0	0	47	*3*	0	0	67	-	-	-	-	C
Apache19101	*1*	0	0	1	*1*	0	0	0	-	-	-	-	Java
Apache32546	*1*	0	0	0	*1*	0	0	0	*135*	24	9	13	Java
Apache34464	*0*	0	0	3	*0*	0	0	2	*1*	0	0	12	Java
Apache44408	*1*	0	1	1	*0*	0	1	2	*3*	1	2	2	Java
Apache45396	*0*	0	0	0	*0*	0	0	1	*0*	0	0	1	Java
Apache48778	*1*	0	0	0	*1*	0	0	0	*19*	14	1	17	Java
Total	*125*	29	41	252	*113*	27	30	191	*219*	49	46	94	

Table 4.2: Checking results. BadPr: bad practice; F.P.: false positives; GoodPr: good practices. More detailed definitions are presented in Section 4.4. '-': not applicable. '/': good-practice checker does not exist.

These results confirm that performance bugs widely exist. Efficiency rules exist and are useful for finding performance problems.

PPPs In Original Versions 17 out of 25 checkers found new PPPs, 125 in total, in the original versions of the buggy software.

Some developers clearly tried to find all similar bugs when fixing one bug, but did not succeed. For example, in MySQL#14637, after two buggy code regions were reported, developers found three more places that were similarly inefficient and fixed them altogether. Unfortunately, there were another 50 code regions that violated the same efficiency rule and skipped developers' checking, as shown in Table 4.2. Similarly, MySQL developers found and fixed

```
//MySQL#15811 & Patch
//strings/ctype-mb.c

- char * end = str + strlen(str);
- if(ismbchar(cs, str, end))
+ if(ismbchar(cs, str, str
                + cs->mbmaxlen))
```

```
//A PPP we found in the latest MySQL
//ibmysql/libmysql.c

//'end' is only used in the ismbchar checking
- for (end=s; *end ; end++) ;
- if (ismbchar(mysqlcs, s, end) )
+ if (ismbchar(mysqlcs, s, s
                + mysqlcs->mbmaxlen)
```

Figure 4.1: A PPP we found in latest versions of *original* software (*ismbchar* checks whether a string (2nd parameter) is coded by a specific character-set (1st parameter). Since *ismbchar* only checks the first CHARSET::mbmaxlen characters of a string, calculating the exact length and range of a string is unnecessary.)

```
//Apache#34464 & Patch
//TelnetTask.java

+ int i = -k.length();
- while (s.indexOf(k) == -1)
+ while (i++<0 ||
+      s.substring(i).indexOf(k)==-1)
  {s.append (nextchar());}
```

```
//A PPP we found in the latest Struts

while (1) {
-  n = s.indexOf("%\\>");
+  n = s.substring(n+2).indexOf("%\\>");
   if (n < 0) break;
   ... // replace "%\\>" by "%>" and continue
}
```

Figure 4.2: A PPP we found in latest versions of *different* software (*String::indexOf* (String *sub*) looks for sub-string *sub* from the beginning of a string s. If program has already compared the first N characters of s with *sub*, it is better not to repeat this. The Struts PPP is already confirmed and patched by Struts developers based on our report.)

3 places that had the inefficiency pattern shown in Figure 4.1, but missed the other 15 places.

113 out of these 125 PPPs exist in different files or even different modules where the original bugs exist, which is probably why they were missed by developers. These PPPs end up in several ways: (1) 4 of them were fixed in later versions, which took 14–31 months; (2) 20 eventually disappeared, because the functions containing these PPPs were removed or re-implemented; (3) 101 still exist in the latest versions of the software, wasting computation resources 12–89 months after the original bugs were fixed.

Lesson The above results show that developers do need support to systematically and automatically find similar performance bugs and fix them all at once.

PPPs In The Latest Versions 2 of the 25 checkers are no longer applicable in the latest versions, because the functions involved in these checkers have been removed. The remaining 23 checkers are applied to the latest versions of corresponding software and find 113 PPPs. Among them, 101 PPPs were inherited from the original buggy versions. The other 12 were introduced later.

Lesson Developers cannot completely avoid the mistakes they made and corrected before, which is understandable considering the large number of bugs in software. Specification systems and automated checkers can prevent developers from introducing old bugs into new code.

PPPs In Different Software Applications An exciting result is that 8 out of 13 cross-application checkers have successfully found previously unknown PPPs in the latest versions of applications that are different from where the rules came from.

Most of these checkers reflect common pitfalls in using library functions. For example, Figure 4.2 shows a pitfall of using `String::indexof()`. Apache-Ant developers made this mistake, and we found Apache-Struts developers also made a similar mistake.

Apache#32546 checker presents an interesting case. In the original bug report, developers from Apache-Slide recognized that a small buffer size would severely hurt the performance of `java.io.InputStream.read (byte buffer[])` for reasonably large input (e.g., larger than 50KB). Replacing their original 2KB buffer with a 200KB buffer achieved **80 times** throughput improvement in WebDav server. We first confirmed that this rule is still valid. Our checker then found 135 places in the latest versions of 36 software applications where similar mistakes were made. These places use small buffers (1KB – 4KB) to read images or data files from disk or web, and are doomed to performance losses.

Some checkers reflect algorithm improvements and are also applicable to many applications. For example, algorithm improvements for string operations

proposed by MySQL developers (MySQL#14637 and MySQL#49491) also apply for Mozilla and Apache HTTPD.

Cross-application checking also helps validate efficiency rules. For example, by comparing how java.util.zip.Deflater.deflate() is used across applications, we found that Ant developers' understanding of this API, reflected by their discussion, was wrong. They fixed Apache#45396 by coincidence.

Lesson The above results show that there exist general inefficiency patterns that go beyond one application, just like that for functional bugs [38]. Maintaining specifications and checkers for these general patterns can significantly save developers' effort, and allow them to learn from other developers and other software. We can even discover performance bugs in a software where no performance patch has ever been filed.

Bad Practices Other than PPPs, some code regions identified by the checkers are categorized as bad practices. For example, there are code regions very similar to the MySQL PPP shown in Figure 4.1, except that the calculation of end is not completely useless as end is used in places other than the invocation of ismbchar. Clearly this practice is more likely to cause performance problems in the future than directly using mysqlcs→mbmaxlen as the parameter for ismbchar function.

Good Practices Code regions that have well followed the efficiency rules are also identified by slightly changed checkers. For example, we found that in 13 places of various applications developers do use InputStream.read (byte buffer[]) in a performance efficient way: buffer has a configurable size or a large size that suits the workload (e.g., 64K in some Hadoop code).

Lesson Violations to efficiency rules are not always rare comparing with good practices. Previous techniques that use statistical analysis to infer functional rules [24, 55] may not work for efficiency rules.

False Positives Our PPP detection is accurate. On average, the false-positive-vs-PPP rate is 1:4. The false positives mainly come from three sources.

First, Python checkers have no object-type information. Therefore, some rules are applied to functions with right function names but wrong classes (e.g., Mozilla#490742 and Apache#32546). This is not a problem in LLVM checkers.

Second, some non-function rules are difficult to accurately express and check, which leads to false positives in MySQL#14637.

Third, accurately checking some efficiency rules requires runtime and/or workload information, which inevitably leads to false positives in our static checkers. False positives in Apache#44408 and Apache#48778 mostly belong to this category. These false positives can be largely eliminated by runtime checkers.

Performance Results Our checkers are efficient. Each Python checker finishes checking 10 million lines of code within 90 seconds. Our LLVM checkers are mainly applied to MySQL, Mozilla Firefox, and Apache HTTPD. It takes 4 – 1270 seconds for one LLVM checker to process one application.

We tried unit testing on PPPs. The performance difference is significant. For example, for programs that read images and files using `InputStream.read(byte buffer[])` with a 4KB-buffer parameter, we can stably get 3 times throughput improvement through a 40K-buffer parameter. When we feed the unit test with a 50MB file, which is a quite common image-file workload these days, the file operation time decreases from 0.87 second to 0.26 second, a definitely perceivable difference. As another example, the Struts code shown in Figure 4.2 is from a utility function used for processing JSP files. Our unit testing with a 15K JSP file shows that the simple patch can decrease latency by 0.1 second, a perceivable difference in interactive web applications.

Whole system testing turns out to be difficult, as suggested by our characteristics study (Section 3.6). No PPP detected by our checkers belongs to the always-active category. Future performance-oriented input-generation tools will significantly help performance testing and identify truly severe PPPs. Execution frequency information can also help future static performance-bug detectors to rank the severity of PPPs.

4.5.1 Discussions

Effectiveness of rule-based performance-bug detection

Effort saving Rule-based detection not only identifies problems, but also suggests alternative implementations with better efficiency. These alternative implementations often have small sizes and regular patterns, as shown in

Figure 4.1 and Figure 4.2, making PPP validation and fixing easy. It is also conceivable to enhance our checkers for automated PPP fixing.

Improving performance These PPPs showed significant performance improvement than their alternative implementations in our unit testing. Without fixing these PPPs, these unit-level performance losses could aggregate into intolerable performance problems that are difficult to diagnose. This is especially significant considering that many performance bugs are difficult to catch using other approaches.

Maintaining code readability Like those 110 patches studied earlier, most PPPs detected by us can be fixed through changes to a few lines of code, as shown in Figure 4.1 and Figure 4.2. Even for the few complicated PPPs, wrapper-functions or macros can easily address the patch-readability issue.

Other usage Rules and checkers can serve as performance specifications for future software development. They can aid in code maintenance when software evolves. Developers can also save PPPs to an inefficiency list for future performance diagnosis.

Of course, this is only a starting point for rule-based performance-bug detection. We expect our experience to motivate future work on automatically generating rules, checkers, or even patches.

Can these problems be detected by other tools?

Copy-paste detectors Most PPPs that we found are **not** from copy-paste code regions and cannot be detected by text-matching tools [28, 53], as we can see in Figure 4.1 and Figure 4.2. Rule violations are not rare. When developers misunderstand an API, they tend to make mistakes whenever they use this API. As a result, these mistakes usually go beyond copy-paste code regions.

Compiler optimization None of the bugs that provided the efficiency rules could be optimized away by compilers used in Apache, MySQL, and Mozilla. Many PPPs involve library functions and algorithmic inefficiency, and are almost impossible for a compiler to optimize (Figure 4.1 and Figure 4.2). Even for the few cases where compiler optimization might help (Figure 3.1), the required inter-procedural and points-to analyses are not scalable for real-world large software.

General rule-based bug detectors Ideas for detecting functional bugs can greatly benefit and inspire future research on performance-bug detection. However, many approaches cannot be directly applied. Tools that automatically infer functional correctness rules [24, 55, 61] may not be suitable for efficiency rules, because rule violations are not rare, as shown in Table 4.2. In addition, many efficiency rules either involve only one function or discourage multiple functions to be used together, making them unsuitable for tools that focus on function correlations.

4.6 Conclusions

Guided by our empirical study in the last chapter, we further explore rule-based performance-bug detection using efficiency rules implied by patches, and find many previously unknown performance problems. Many reported performance problems have already confirmed and fixed by the developers. Our work shows that rule-based performance-bug detection is promising.

Chapter 5

Statistical Debugging for Real-World Performance Bugs

As we discussed in previous chapters, performance bugs are difficult to avoid due the lack of performance documentation and quickly changing workloads. Many performance bugs escape from in-house testing process and manifest in front of end users. Effective tools that diagnose performance problems and point out the inefficiency root cause are sorely needed.

The state of the art of performance diagnosis is preliminary. Profiling can identify the functions that consume the most computation resources, but can neither identify the ones that waste the most resources nor explain why. Performance-bug detectors can identify specific type of inefficient computation, but are not suited for diagnosing general performance problems. Effective failure diagnosis techniques, such as statistical debugging, have been proposed for functional bugs. However, whether they work for performance problems is still an open question.

In this chapter, we first conduct an empirical study to understand how performance problems are observed and reported by real-world users. Our study shows that statistical debugging is a natural fit for diagnosing performance problems, which are often observed through comparison-based approaches and reported together with both good and bad inputs. We then thoroughly investigate different design points in statistical debugging, including three different predicates and two different types of statistical models, to understand which design point works the best for performance diagnosis. Finally, we study

how some unique nature of performance bugs allows sampling techniques to lower the overhead of runtime performance diagnosis without extending the diagnosis latency.

5.1 Introduction

5.1.1 Motivation

As we discussed in Chapter 3, performance bugs are difficult to avoid during implementation, and are also difficult to expose during in-house testing. Many performance bugs manifest in front of end users and severely hurt users' experience during production runs. After users report performance bugs, developers need to quickly diagnose them and fix them. Diagnosing user-reported performance bugs is one important aspect to combat performance bugs. Effective tool support is sorely needed.

The state of practice of performance diagnosis is preliminary. The most commonly used and often the only available tool during diagnosis is profiler [1, 77]. Although useful, profilers are far from sufficient. They can tell where computation resources are spent, but not where or *why* computation resources are *wasted*. As a result, they still demand a huge amount of manual effort to figure out the root cause[1] of performance problems.

Figure 3.6 shows a real-world performance problem in MySQL. MySQL users noticed surprisingly poor performance for queries on certain type of tables. Profiling could not provide any useful information, as the top ranked functions are either low-level library functions, like `pthread_getspecific` and `pthread_mutex_lock`, or simple utility functions, like `ha_key_cmp` (key comparison). After thorough code inspection, developers finally figured out that the problem is in function `start_bulk_insert`, which does not even get ranked by the profiler. The developer who implemented this function assumed that parameter-0 indicates no need of cache, while the developers who wrote the caller functions thought that parameter-0 indicates the allocation of a large buffer. This mis-communication led to unexpected cache-less execution, which

[1] Root cause refers to a static code region that can cause inefficient execution.

is extremely slow. The final patch simply removes the unnecessary branch in Figure 3.6, but it took developers a lot of effort to figure out.

Most recently, non-profiling tools have been proposed to help diagnose certain type of performance problems. For example, X-Ray can help pin-point the configuration entry or input entry that is most responsible for poor performance [6]; trace analysis techniques have been proposed to figure out the performance-causality relationship among system events and components [21, 106]. Although promising, these tools are still far from automatically identifying source-code level root causes and helping figure out source-code level fix strategies for general performance problems.

Many automated performance-bug detection tools have been proposed recently, but they are ill suited for performance diagnosis. Each of these tools detects one specific type of performance bugs, such as inefficient nested loops [75], under-utilized data structures [102], and temporary object bloat [22, 100, 103], through static or dynamic program analysis. They are not designed to cover a wide variety of performance bugs. They are also not designed to focus on any specific performance symptom reported by end users, and would inevitably lead to false positives when used for failure diagnosis.

5.1.2 Can we learn from functional failure diagnosis?

Automated failure diagnosis has been studied for decades for functional bugs Many useful and generic techniques [34, 37, 43, 45, 56, 109] have been proposed. Among these techniques, statistical debugging is one of the most effective [43, 45, 56]. Specifically, statistical debugging collects program predicates, such as whether a branch is taken, during both success runs and failure runs, and then uses statistical models to automatically identify predicates that are most correlated with a failure, referred to as failure predictors. It would be nice if statistical debugging can also work for diagnosing performance problems.

Whether statistical debugging is useful for performance bugs is still an open question. Whether it is *feasible* to apply the statistical debugging technique to performance problems is unclear, not to mention *how* to apply the technique.

Is it feasible to apply statistical debugging? The prerequisites for statistical debugging are two sets of inputs, one leading to success runs, referred to as

good inputs, and one leading to failure runs, referred to as *bad inputs*. They are easy to obtain for functional bugs, but may be difficult for some performance bugs.

For functional bugs, failure runs are often easy to tell from success runs due to clear-cut failure symptoms, such as crashes, assertion violations, incorrect outputs, and hangs. Consequently, it is straightforward to collect good and bad inputs. In the past, the main research challenge has been generating good inputs and bad inputs that are similar with each other [109], which can improve the diagnosis quality.

For some performance bugs, failure runs could be difficult to distinguish from success runs, because execution slowness can be caused by either large workload or manifestation of performance bugs.

Empirical study is needed to understand whether statistical debugging is feasible for real-world performance bugs and, if feasible, how to obtain good inputs and bad inputs.

How to conduct effective statistical debugging? The effectiveness of statistical debugging is not guaranteed by the availability of good and bad inputs. Instead, it requires careful design of predicates and statistical models that are suitable for the problem under diagnosis.

Different predicates and statistical models have been designed to target different types of common functional bugs. For example, branch predicates and function-return predicates have been designed to diagnose sequential bugs [56, 57]; interleaving-related predicates have been designed to diagnose concurrency bugs [5, 43]; ΔLDA statistical model [4] has been used to locate failure root causes that have weak signals. What type of predicates and statistical models, if any, would work well for performance diagnosis is still an open question.

5.1.3 Contributions

This chapter presents a thorough study of statistical debugging for real-world performance problems. Specifically, it makes the following contributions.

An empirical study of the diagnosis process of real-world user-reported performance problems To understand whether it is feasible to apply statisti-

cal debugging for real-world performance problems, we study how users notice and report performance problems based on 65 real-world user-reported performance problems in five representative open-source applications (Apache, Chrome, GCC, Mozilla, and MySQL). We find that statistical debugging is feasible for most user-reported performance problems in our study, because (1) users notice the symptoms of most performance problems through a comparison-based approach (more than 80% of the cases), and (2) many users report performance bugs together with two sets of inputs that look similar with each other but lead to huge performance difference (about 60% of the cases). Furthermore, we also find that performance diagnosis is time consuming, taking more than 100 days on average, and lacking good tool support, taking more than 100 days on average even after profiling. Although our work is far from a full-blown study of all real-world user-reported performance bugs, its findings still provide guidance and motivation for statistical debugging on performance problems. The details are in Section 5.2.

A thorough study of statistical in-house performance diagnosis To understand how to conduct effective statistical debugging for real-world performance problems, we set up a statistical debugging framework and evaluate a set of design points for user-reported performance problems. These design points include three representative predicates (branches, function returns, and scalar-pairs) and two different types of statistical models. They are evaluated through experiments on 20 user-reported performance problems and manual inspections on all the 65 user-reported performance problems collected in our empirical study. Our evaluation demonstrates that, when the right design points are chosen, statistical debugging can effectively provide root-cause and fix-strategy information for most real-world performance problems, improving the state of the art of performance diagnosis. More details are presented in Section 5.3.

A thorough study of sampling-based production-run performance diagnosis We apply both hardware-based and software-based sampling techniques to lower the overhead of statistical performance diagnosis. Our evaluation using 20 real-world performance problems shows that sampling does not

degrade the diagnosis capability, while effectively lowering the overhead to below 10%. We also find that the special nature of loop-related performance problems allows the sampling approach to lower runtime overhead without extending the diagnosis latency, a feat that is almost impossible to achieve for sampling-based functional-bug failure diagnosis. More details are presented in Section 5.4.

5.2 Understanding Real-World Performance Problem Reporting and Diagnosis

This section aims to understand the performance diagnosis process in real world. Specifically, we will focus on these two aspects of performance diagnosis.

1. How users notice and report performance problems. This will help us understand the feasibility of applying statistical debugging to real-world performance problems, as discussed in Section 5.1.2. Particularly, we will study how users tell success runs from failure runs in the context of performance bugs and how to obtain success-run inputs (i.e., good inputs) and failure-run inputs (i.e., bad inputs) for performance diagnosis.

2. How developers diagnose performance problems. This will help us understand the state of practice of performance diagnosis.

5.2.1 Methodology

The performance problems under this study include all user-reported performance problems from the benchmark suite collected in Chapter 3. We cannot directly use the baseline benchmark suite, because it contains bugs that are discovered by developers themselves through code inspection, a scenario that performance diagnosis does not apply. Consequently, we carefully read through all the bug reports and identify all the **65** bugs that are clearly reported by users. These 65 bug reports all contain detailed information about how each performance problem is observed by a user and gets diagnosed by developers. They are the target of the following characteristics study, and will be referred to as *user-reported performance problems* or simply *performance problems* in the

Application Suite Description (language)	# Bugs
Apache Suite HTTPD: Web Server (C) TomCat: Web Application Server (Java) Ant: Build management utility (Java)	16
Chromium Suite Google Chrome browser (C/C++)	5
GCC Suite GCC & G++ Compiler (C/C++)	9
Mozilla Suite Firefox: Web Browser (C++, JavaScript) Thunderbird: Email Client (C++, JavaScript)	19
MySQL Suite Server: Database Server (C/C++) Connector: DB Client Libraries (C/C++/Java/.Net)	16
Total	65

Table 5.1: Applications and bugs used in the study.

remainder of this chapter. The detailed distribution of these 65 bugs is shown in Table 5.1.

Caveats Similar with all previous characteristics studies, our findings and conclusions need to be considered with our methodology in mind. As discussed in Chapter 3, the bugs in our study are collected from representative applications without bias. We have followed users and developers' discussion to decide what are performance problems that are noticed and reported by users, and finally diagnosed and fixed by developers. We did not intentionally ignore any aspect of performance problems. Of course, our study does not cover performance problems that are not reported to or fixed in the bug databases. It also does not cover performance problems that are indeed reported by users but have undocumented discovery and diagnosis histories. Unfortunately, there is no conceivable way to solve these problems. We believe the bugs in our study provide a representative sample of the well-documented fixed performance bugs that are reported by users in representative applications.

5.2.2 How users report performance problems

In general, to conduct software failure diagnosis, it is critical to understand what are the failure symptoms and what information is available for failure

Categories	Apache	Chrome	GCC	Mozilla	MySQL	Total
Comparison within one code base	9	3	7	7	12	38
Comparing the same input with different configurations	2	1	1	1	5	10
Comparing inputs with different sizes	6	2	4	4	6	22
Comparing inputs with slightly different functionality	2	0	3	2	4	11
Comparison cross multiple code bases	7	3	8	5	4	27
Comparing the same input under same application's different versions	4	2	8	3	3	20
Comparing the same input under different applications	4	1	1	2	1	9
Not using comparison-based methods	3	1	0	9	1	14

Table 5.2: How performance problems are observed by end users. There are overlaps among different comparison-based categories; there is no overlap between non-comparison and comparison-based categories.

diagnosis. Specifically, as discussed in Section 5.1.2, to understand the feasibility of applying statistical debugging for performance diagnosis, we will investigate two issues: (1) How do users judge whether a slow execution is caused by large workload or inefficient implementation, telling success runs from failure runs? (2) What information do users provide to convince developers that inefficient implementation exists and hence help the performance diagnosis?

How are performance problems observed? As shown in Table 5.2, the majority (51 out of 65) of user-reported performance problems are observed through comparison, including comparisons within one software code base and comparisons across multiple code bases.

Comparison within one code base is the most common way to reveal performance problems. In about 60% of cases, users notice huge performance differences among similar inputs and hence file bug reports.

Sometimes, the inputs under comparison have the same functionality but different sizes. For example, MySQL#44723 is reported when users observe that inserting 11 rows of data for 9 times is two times slower than inserting 9 rows of data for 11 times. As another example, Mozilla#104328 is reported when users observe a super-linear performance degradation of the web-browser start-up time in terms of the number of bookmarks.

Sometimes, the inputs under comparison are doing slightly different tasks. For example, when reporting Mozilla#499447, the user mentions that changing the width of Firefox window, with a specific webpage open, takes a lot of time (a bad input), yet changing the height of Firefox window, with the same webpage, takes little time (a good input).

Finally, large performance difference under the same input and different configurations is also a common reason for users to file bug reports. For example, when reporting GCC#34400, the user compared the compilation time of the same file under two slightly different GCC configurations. The only difference between these two configurations is that the "ZCX_By_Default" entry in the configuration file is switched from True to False. However, the compilation times goes from 4 seconds to almost 300 minutes.

Comparison across different code bases In about 40% of the performance problems that we studied, users support their performance suspicion through a comparison across different code bases. For example, GCC#12322 bug report mentions that "GCC-3.3 compiles this file in about five minutes; GCC-3.4 takes 30 or more minutes". As another example, Mozilla#515287 bug report mentions that the same Gmail instance leads to 15–20% CPU utilization in Mozilla Firefox and only 1.5% CPU utilization in Safari.

Note that, the above two comparison approaches do not exclude each other. In 14 out of 27 cases, comparison results across multiple code bases are reported together with comparison results within one code base.

Non-comparison based For about 20% of user-reported performance problems, users observe an absolutely non-tolerable performance and file the bug report without any comparison. For example, Mozilla#299742 is reported as the web-browser frozed to crawl.

What information is provided for diagnosis? The most useful information provided by users include failure symptom (discussed above), bad inputs, and good inputs. Here, we refer to the inputs that lead to user-observed performance problems as *bad inputs*; we refer to the inputs that look similar with some bad inputs but lead to good performance, according to the users, as *good inputs*.

	Apache	Chrome	GCC	Mozilla	MySQL	Total
Total # of bug reports	16	5	9	19	16	65
# of bad inputs provided						
0/?: No bad input	0	0	0	0	0	0
1/?: One bad input	0	1	5	6	7	19
n/?: A set of bad inputs	16	4	4	13	9	46
# of good inputs						
?/0: No good input	7	2	2	12	4	27
?/1: One good input	0	0	3	0	3	6
?/n: A set of good inputs	9	3	4	7	9	32

Table 5.3: Inputs provided in users' bug reports. n: developers provide a way to generate a large number of inputs.

Bad inputs Not surprisingly, users provide problem-triggering inputs in all the 65 cases. What is interesting is that in about 70% of cases (46 out of 65), users describe a category of inputs, instead of just one input, that can trigger the performance problem, as shown in Table 5.3. For example, in MySQL#26527, the user describes that loading data from file into partitioned table can trigger the performance problem, no matter what is the content or schema of the table.

Good inputs Interestingly, good inputs are specified in almost 60% of bug reports, as shown in Table 5.3. That is, users describe inputs that look similar with the bad inputs but have much better performance in all the 38 bug reports where "comparison within one code base" is used to observe the performance problem. Furthermore, in 32 bug reports, users describe how to generate a large number of good inputs, instead of just one good input. For example, when reporting MySQL#42649, the user describes that executing queries on tables using the default charset setting or the *latin1* charset setting (good inputs) will not cause lock contention, while queries on tables using other types of charset settings (bad inputs) may cause lock contention. Note that, this is much rarer in functional bug reports, which is why special tools are designed to automatically generate inputs that execute correctly and are similar with bad inputs, when diagnosing functional-bug failures [109].

5.2.3 How developers diagnose performance problems

To collect the diagnosis time, we check the bug databases and calculate the time between a bug report being posted and a correct fix being proposed. Of course, strictly speaking, this time period can be further broken down to bug-report assignment, root-cause locating, patch design, and so on. Unfortunately, we cannot obtain such fine-grained information accurately from the databases. Most Apache, Chrome, and MySQL bugs in our study do not have clear assignment time in record. For GCC bugs in study, report assignment takes about 1% of the overall diagnosis time on average; for Mozilla bugs in study, report assignment takes about 19% of the overall diagnosis time on average.

Our study shows that it takes 129 days on average for developers to finish diagnosing a performance problem reported by users. Among the 5 software projects, the Chrome project has the shortest average performance-diagnosis time (59 days), and Apache project has the longest average diagnosis time (194 days). Comparing with the numbers reported in Chapter 3, the time to diagnose user-reported performance problems is slightly shorter than that for non-user-reported performance problems, and similar or longer than that of functional bugs.

We also studied how developers diagnose performance problems. The only type of diagnosis tools that are mentioned in bug reports are performance profilers. They are mentioned in 13 out of the 65 reports. However, even after the profiling results are provided, it still takes developers 116 days on average to figure out the patches.

5.2.4 Implications of the study

Implication 1 Performance bugs and functional bugs are observed in different ways. Intuitively, the symptoms of many functional bugs, such as assertion violations, error messages, and crashes, can be easily identified by looking at the failure run alone [54]. In contrast, the manifestation of performance bugs often gets noticed through comparison. We have randomly sampled 65 user-reported functional bugs from the same set of applications (i.e., Apache, Chrome, GCC, Mozilla, and MySQL) and found that only 8 of them are observed through comparison. Statistical Z tests [96] show that the above observation is statisti-

cally significant — at the 99% confidence level, a user-reported performance bug is more likely to be observed through comparison than a user-reported functional bug.

Implication 2 Although judging execution efficiency based on execution time alone is difficult in general, distinguishing failure runs from success runs and obtaining bad and good inputs are fairly straightforward based on performance-bug reports filed by users. Our study shows that most user-reported performance problems are observed when two sets of similar inputs demonstrate very different performances (38 out of 65 cases). Most of these cases (32 out of 38), users provide explicit good and bad input-generation methodology. In other cases (27 out of 65), users observe that an input causes intolerably slow execution or very different performances across similar code bases. Distinguishing failure runs from success runs and bad inputs from good inputs are straightforward in these cases based on the symptoms described in the bug reports, such as "frozed the GUI to crawl" in Mozilla#299742 and 10X more CPU utilization rate than Safari under the same input in Mozilla#515287.

Implication 3 Statistical debugging is naturally suitable for diagnosing many user-reported performance problems, because most performance bugs are observed by users through comparison and many performance-bug reports (38 out of 65) already contain information about both bad and good inputs that are similar with each other. Statistical tests [96] show that with 90% statistical confidence, a user-filed performance-bug report is more likely to contain both bad and good inputs than not. Comparing the 65 randomly sampled functional bugs mentioned above with the 65 performance bugs, statistical tests [96] show that, at the 99% confidence level, a user-filed performance-bug report is more likely to contain good inputs than a user-filed functional-bug report. Previous statistical debugging work tries hard to generate good inputs to diagnose functional bugs [109]. This task is likely easier for performance failure diagnosis.

Implication 4 Developers need tools, in addition to profilers, to diagnose user-reported performance problems.

5.3 In-house Statistical Debugging

During in-house performance diagnosis, users send detailed bug reports to the developers and developers often repeat the performance problems observed by the users before they start debugging. Following the study in Section 5.2, this section designs and evaluates statistical debugging for in-house diagnosis of real-world performance problems. We aim to answer three key questions.

1. What statistical debugging design is most suitable for diagnosing real-world performance problems;
2. What type of performance problems can be diagnosed by statistical debugging;
3. What type of performance problems cannot be diagnosed by statistical debugging alone.

5.3.1 Design

In general, statistical debugging [3, 5, 43, 45, 56, 57, 86] is an approach that uses statistical machine learning techniques to help failure diagnosis. It usually works in two steps. First, a set of runtime events E are collected from both success runs and failure runs. Second, a statistical model is applied to identify an event $e \in E$ that is most correlated with the failure, referred to as the failure predictor. Effective statistical debugging can identify failure predictors that are highly related to failure root causes and help developers fix the underlying software defects.

There are three key questions in the design of statistical debugging.

1. Input design – what inputs shall we use to drive the incorrect execution and the correct execution during statistical debugging. If the good runs and the bad runs are completely different (e.g., they do not cover any common code regions), the diagnosis will be difficult.
2. Predicate design – what type of runtime events shall we monitor. Roughly speaking, a predicate P_i could be true or false, depending on whether a

specific property is satisfied at instruction i at run time. To support effective diagnosis, one should choose predicates that can reflect common failure root causes.

3. Statistical model design – what statistical model shall we use to rank predicates and identify the best failure predictors among them.

The input design problem is naturally solved for performance diagnosis, as discussed in Section 5.2. We discuss different predicate designs and statistical model designs below.

5.3.1.1 Predicate designs

Many predicates have been designed to diagnose functional bugs. We discuss some commonly used ones below.

Branches. There are two branch predicates associated with each branch b: one is true when b is taken, and the other is true when b is not taken [56, 57].

Returns. There are a set of six return predicates for each function return point, tracking whether the return value is ever < 0, $\leqslant 0$, > 0, $\geqslant 0$, $= 0$, or $\neq 0$ [56, 57].

Scalar-pairs. There are six scalar-pair predicates for each pair of variables x and y, tracking whether x is ever $< y$, $\leqslant y$, $> y$, $\geqslant y$, $= y$, or $\neq y$ [56, 57]. Whenever a scalar variable x is updated, scalar-pair predicates are evaluated between x and each other same-type variable y that is in scope, as well as program constants.

Instructions. Instruction predicate i is true, if i has been executed during the monitored run [3, 45, 86].

Interleaving-related ones. Previous work on diagnosing concurrency bugs [43] has designed three types of predicates that are related to thread interleaving. For example, CCI-Prev predicates track whether two consecutive accesses to a shared variable come from two distinct threads or the same thread.

In the remainder of this section, we will focus on three predicates: branch predicates, return predicates, and scalar-pair predicates. We skip instruction predicates in this study, because they are highly related to branch predicates.

BugID	KLOC	Language	Static # of predicates			Static # of Loops	Reported Inputs (bad/good)
			Branch	Return	Scalar-pair		
Mozilla258793	3482	C++	385722	1126770	*	10016	n/0
Mozilla299742	3482	C++	385720	1126698	*	10016	1/0
Mozilla347306	88	C	26804	38634	271968	951	n/n
Mozilla416628	105	C	28788	39306	302496	1420	1/0
MySQL15811	1149	C++	13508	15576	*	760	n/n
MySQL26527	986	C++	90128	128610	*	4222	n/n
MySQL27287	995	C++	92316	119322	*	4683	n/n
MySQL40337	1191	C++	103686	138582	*	4510	n/n
MySQL42649	1164	C++	126822	155766	*	5688	n/n
MySQL44723	1164	C++	126822	155766	*	5688	1/1
Apache3278	N/A	Java	10	126	204	7	n/n
Apache34464	N/A	Java	22	42	342	8	n/n
Apache47223	N/A	Java	24	36	390	9	n/n
Apache32546	N/A	Java	6	66	120	5	n/n
GCC1687	2099	C	183496	296058	4187586	6476	n/n
GCC8805	2538	C	207188	327804	4161012	7309	n/n
GCC15209	2586	C	192108	304800	3705558	7310	1/1
GCC21430	3844	C	238514	447510	3768078	9078	n/n
GCC46401	5521	C	337810	713532	5625606	15159	1/1
GCC12322	2341	C	177098	284484	3750912	6563	1/0

Table 5.4: Benchmark information. N/A: since our statistical debugging tools only work for C/C++ programs, we have reimplemented the four Java benchmarks in C programs. *: we have no tools to collect scalar-pair predicates in C++ programs. The 1s and ns in the "Reported Inputs" column indicate how many bad/good inputs are reported by users.

We skip interleaving-related predicates in this study, because most performance problems that we study are deterministic and cannot be effectively diagnosed by interleaving-related predicates.

5.3.1.2 Statistical model designs

Many statistical models have been used before for anomaly detection [24, 36, 51, 53] and fault localization [3, 4, 43, 45, 56, 57, 86]. Although the exact models used by previous work differ from each other, they mostly follow the same principle — if a predicate is a good failure predictor, it should be true in many failure runs, and be false or not-observed in many success runs. They can be roughly categorized into two classes. The first class only considers whether a predicate has been observed true for at least once in a run (e.g., whether

a branch b has been taken for at least once). The exact number of times the predicate has been true in each run is not considered in the model. The second class instead considers the exact number of times a predicate has been true in each run. Naturally, by considering more information in the model, the second class could complement the first class, but at the cost of longer processing time. Most previous work on functional bug diagnosis has found the first class sufficient [5, 43, 56, 57] and did not try the second class.

To cover both classes of statistical models for performance diagnosis, our study will look at two models: a *basic* model proposed by CBI work [56, 57] that belongs to the first class discussed above and a ΔLDA model proposed by [4] that belongs to the second class discussed above. We leave investigating other existing statistical models and designing new models to future work. Since our evaluation will use exactly the same formulas, parameters, and settings for these two models as previous work [4, 56, 57], we briefly discuss these two models below. More details about these two models can be found in their original papers [4, 56, 57].

Basic model This model works in two steps. First, it checks whether an execution is more likely to fail when a predicate P is observed true, whose probability is computed by formula *Failure*(P), than when P has merely being observed during the execution, whose probability is computed by formula *Context*(P). Consequently, only predicates, whose *Increase* values computed below are higher than 0 with certain statistical confidence, will appear in the final ranking list. By default, statistical Z-tests and 0.99 confidence level are used in CBI [56].

$$Failure(P) = \frac{F(P_{true})}{S(P_{true}) + F(P_{true})}$$

$$Context(P) = \frac{F(P_{observed})}{S(P_{observed}) + F(P_{observed})}$$

$$Increase(P) = Failure(P) - Context(P)$$

$F(P_{true})$ is the number of failure runs in which P is true, and $F(P_{observed})$ is the number of failure runs in which P is observed, no matter true or false. $S(P_{true})$ is the number of success runs in which P is true, and $S(P_{observed})$ is the number of success runs in which P is observed.

$$Importance(P) = \frac{2}{\frac{1}{Increase(P)} + \frac{1}{\log(F(P_{true}))/\log(F)}}$$

The final ranking is based on an *Importance* metric. This metric reflects the harmonic mean of the *Increase* metric and the conditional probability of a predicate P being true given that an execution has failed. F is the total number of failure runs in the formula above. Previous work [57] has tried different variants of the harmonic mean and found the formula above, with a logarithmic transformation, to be the best. As mentioned above, we reuse all the formulas, parameters, and settings from previous work.

ΔLDA model ΔLDA [4] model is derived from a famous machine learning model, called Latent Dirichlet Allocation (LDA) [8]. By considering how many times a predicate is true in each run, it can pick up weaker bug signals, as shown by previous work [4]. Imagine the following scenario — during a success run, predicate P is true for 10 times and false for 100 times; during a failure run, P is true for 100 times and false for 10 times. The basic model will consider P as useless, as it has been observed both true and false in every run. However, ΔLDA model will notice that P is true for many more times during each failure run than that in each success run, and hence consider P as failure predictor. The exact ranking formula of ΔLDA model is very complicated, and is skipped here. It can be found in previous work [4].

How to apply the models A statistical debugging framework collects the following information from each run: (1) whether the run has succeeded and failed; (2) a list of predicates that have been observed true and for how many times each (the latter only for ΔLDA model). After collecting such information from several success runs and failure runs, the framework will naturally obtain values, such as the number of failure runs where a predicate is observed/true, for the formulas discussed above and produce a rank list of failure predictors.

5.3.2 Experimental evaluation

5.3.2.1 Methodology

To evaluate how statistical debugging works for real-world performance problems, we apply three types of predicates and two types of statistical models on real-world user-reported performance problems. All our experiments are conducted on an Intel i7-4500U machine, with Linux 3.11 kernel.

Benchmark selection Among the 65 user-reported performance problems discussed in Section 5.2, we have tried our best effort and successfully repeated 20 of them from four different C/C++/Java applications. In fact, most of the 65 performance problems are deterministically repeatable based on the bug reports. We have failed to repeat 45 of them for this study mainly because they depend on special hardware platforms or very old libraries that are not available to us or very difficult to set up. The detailed information for the 20 performance problems used in our experiments is shown in Table 5.4. Specifically, the static number of branch predicates is counted based on the fact that there are two predicates for each static branch instruction in the user program (excluding library code). The static numbers of other predicates are similarly counted.

To make sure these 20 benchmarks are representative, we also conduct manual source-code inspection to see how statistical debugging could work for **all** the 65 user-reported performance problems in our study. We will show that our manual inspection results on all the 65 cases are **consistent** with our experimental evaluation on these 20 benchmarks.

Input design To conduct the statistical debugging, we run each benchmark program 20 times, using 10 unique good inputs and 10 unique bad inputs. For each performance problem, we get its corresponding 20 inputs based on users' bug report. For 13 of them, the bug reports have described how to generate a large number of good and bad inputs, which makes our input generation straightforward. For the remaining 7 bugs, with 3 from Mozilla, 3 from GCC, and 1 from MySQL, we randomly change the provided inputs and use the user-provided failure-symptom information to decide which inputs are good or bad. We make sure that inputs generated by us are still valid HTML webpages,

valid JavaScript programs, valid C programs, or valid database tables/queries. The process of judging which inputs are good or bad is straightforward, as discussed in Section 5.2.4. For example, Mozilla#299742 reports a webpage that leads to a consistent CPU usage rate above 70%, while some similar webpages lead to less than 10% CPU usage rate. We generate many inputs by randomly replacing some content of this webpage with content from other randomly picked webpages, and judge whether the inputs are good or bad based on CPU usage.

Techniques under comparison We will evaluate three predicates (branches, returns, scalar-pairs) and two statistical models (basic, ΔLDA) for statistical debugging. For C programs, we use CBI [56, 57] to collect all these three types of predicates[2]. For C++ programs, we implement our own branch-predicate and return-predicate collection tools using PIN binary-instrumentation framework [64]. Scalar-pair predicates are very difficult to evaluate using PIN, and hence are skipped for C++ programs in our experimental evaluations. They will be considered for all benchmarks in our manual study (Section 5.3.3). For Java programs, we re-implement them by using C, and use CBI to collect all the three types of predicates. Since the exact execution time is not the target of our information collection, we did not encounter any observer effect in our experiment.

We use the default settings of the CBI basic model and the ΔLDA model for *all* the benchmarks in our evaluation. Specifically, CBI model only has one parameter — the statistical confidence level for filtering out predicates based on the *Increase* metric. We use the default setting 0.99. The key parameter in ΔLDA model is the number of bad topics. We use the default setting 1.

We also use *OProfile* [77] to get profiling results in our experiments. We provide two types of profiling results, both of which are under the "Profiler" column in Table 5.5. "Fun" demonstrates where the root-cause function ranks in the profiler result and what is the distance between the root-cause function and where patches are applied. "Stack" considers the call-chain information

[2] CBI [56, 57] is a C framework for lightweight instrumentation and statistical debugging. It collects predicate information from both success and failure runs, and utilize statistical model to identify the likely causes of software failures.

	# of candidate predicates			Basic model			ΔLDA model			Profiler	
BugID	Branch	Return	S-pair	Branch	Return	S-pair	$\text{Branch}_{\text{loop}}$	Return	S-pair	Fun	Stack
Mozilla258793	62822	149354	*	$\checkmark_1(0)$	-	*	-	-	*	-	-
Mozilla299742	61256	148688	*	$\checkmark_1(0)$	-	*	-	-	*	-	-
Mozilla347306	3931	4062	21590	-	-	-	$\checkmark_1(1)$	$\checkmark_1(1)$	$\checkmark_1(1)$	$\checkmark_1(7)$	$\checkmark_{1[0]}$
Mozilla416628	3719	3598	19428	-	-	-	$\checkmark_1(.)$	-	$\checkmark_1(.)$	$\checkmark_1(.)$	$\checkmark_{1[0]}$
MySQL15811	1198	866	*	-	-	*	$\checkmark_1(.)$	$\checkmark_1(0)$	*	$\checkmark_1(.)$	$\checkmark_{1[0]}$
MySQL26527	6422	6823	*	$\checkmark_1(0)$	-	*	-	-	*	-	-
MySQL27287	5377	5752	*	-	-	*	$\checkmark_1(0)$	-	*	$\checkmark_1(0)$	$\checkmark_{1[0]}$
MySQL40337	7868	8160	*	$\checkmark_1(1)$	-	*	-	-	*	-	-
MySQL42649	12569	9696	*	$\checkmark_1(.)$	-	*	-	-	*	-	-
MySQL44723	10476	9108	*	$\checkmark_1(.)$	-	*	-	-	*	-	$\checkmark_{1[2]}$
Apache3278	7	63	102	$\checkmark_1(3)$	$\checkmark_1(2)$	$\checkmark_1(2)$	-	-	-	-	-
Apache34464	17	23	193	-	-	-	$\checkmark_3(0)$	$\checkmark_1(2)$	-	$\checkmark_5(2)$	$\checkmark_{1[1]}$
Apache47223	17	15	237	-	-	-	$\checkmark_1(.)$	-	$\checkmark_1(.)$	$\checkmark_1(.)$	$\checkmark_{1[0]}$
Apache32546	5	34	69	-	-	-	$\checkmark_1(8)$	$\checkmark_1(7)$	$\checkmark_1(7)$	-	$\checkmark_{5[0]}$
GCC1687	22602	17787	428103	-	-	-	$\checkmark_1(.)$	$\checkmark_2(.)$	-	$\checkmark_1(.)$	$\checkmark_{1[0]}$
GCC8805	23891	20467	404594	-	-	-	$\checkmark_4(0)$	$\checkmark_1(0)$	-	-	-
GCC15209	8956	9403	155007	$\checkmark_1(13)$	-	-	-	-	-	-	-
GCC21430	45494	51270	647228	-	-	-	$\checkmark_1(0)$	-	$\checkmark_1(0)$	$\checkmark_1(2)$	$\checkmark_{1[0]}$
GCC46401	34365	38263	479508	-	-	-	$\checkmark_2(.)$	$\checkmark_3(.)$	$\checkmark_1(.)$	$\checkmark_5(.)$	$\checkmark_{1[2]}$
GCC12322	46721	38269	878823	-	-	-	-	-	-	-	$\checkmark_{1[1]}$

Table 5.5: Experimental results for in-house diagnosis. $\checkmark_x(y)$: the x-th ranked failure predictor is highly related to the root cause, and is y lines of code away from the patch. $(.)$: the failure predictor and the patch are more than 50 lines of code away from each other or are from different files. $\checkmark_{x[y]}$: a y-th level caller of the x-th ranked function in a profiler result is related to the root cause; $_x[0]$ means it is the function itself that is related to the root cause. -: none of the top five predictors are related to the root cause or no predicates reach the threshold of the statistical model.

provided by OProfile for each function in its ranking list. It first checks whether any direct or indirect caller functions of the top OProfile-ranked function is related to the root cause; if not, it then checks the callers, callers' callers, and so on of the second top ranked function; and so on. Among the 65 bug reports in our study, 13 of them mentioned the use of profilers. Among these 13, 4 of them mentioned the use of call-chain information provided by the profilers. For the simplicity of explanation, we will use the "Fun" setting as the default setting for discussing profiler results, unless specified otherwise.

```
1  notified = false;
2  while(!notified) {
3    rc = pthread_cond_timedwait(
4              &cond, &lock, &timeToWait);
5    if(rc == ETIMEDOUT) {
6      break;
7    }
8  }
```

Figure 5.1: An Apache bug diagnosed by Return

5.3.2.2 Results for basic model

Overall, 8 out of 20 performance problems can be successfully diagnosed using the basic statistical model. Furthermore, in all these 8 cases, the failure predictor that is ranked number one by the statistical model is indeed highly related to the root cause of the performance problem. Consequently, developers will not waste their time in investigating spurious failure predictors.

Among all three types of evaluated predicates, the branch predicate is the most useful, successfully diagnosing 8 benchmarks.

The scalar-pair predicate and function-return predicate are only useful for diagnosing one performance problem, as shown in Figure 5.1. In Apache#3278, users describe that Tomcat could non-deterministically take about five seconds to shut-down, which is usually instantaneous. When applied to Tomcat executions with fast and slow shut-downs, statistical debugging points out that there are strong failure predictors from all three types of predicates — (1) the `if(rc==ETIMEDOUT)` branch on line 5 being taken (branch predicate); (2) the `pthread_cond_timedwait` function returning a positive value (function-return predicate); (3) the value of `rc` on line 3 after the assignment is larger than its original value before the assignment (scalar-pair predicate)[3]. These three predicates actually all indicate that `pthread_cond_timedwait` times out without getting any signal. A closer look at that code region shows that developers initialize `notified` too late. As a result, another thread could set `notified` to be `true` and issue a signal even before the `notified` is initialized to be `false`

[3]CBI does not consider program constants for its scalar-pair predicates by default, and hence cannot capture the comparison between `rc` and `ETIMEDOUT` here.

```
//ha_myisam.cc
/* don't enable row cache if too few rows */
if (! rows || (rows > MI_MIN_ROWS_TO_USE_WRITE_CACHE) )
    mi_extra(...);
//mi_extra() will allocate write cache
//and zero-fill write cache
// fix is to remove zero-fill operation
....
// in myisamdef.h:
// #define MI_MIN_ROWS_TO_USE_WRITE_CACHE 10
```

Figure 5.2: A MySQL bug diagnosed by Branch

on line 1 of Figure 5.1, causing a time-out in `pthread_cond_timedwait`. This problem can be fixed by moving `notified=false;` earlier.

In most cases, the failure predictor is very close to the final patch of the performance problem (within 10 lines of code). For example, the patch for the Apache bug in Figure 5.1 is only two lines away from the failure predictor. As another example, the top-ranked failure predictor for the MySQL bug shown in Figure 3.6 is at the **if**(!rows) branch, and the patch exactly changes that branch.

There are also two cases, where the failure predictor is highly related to the root cause but is in different files from the final patch. For example, Figure 5.2 illustrates the performance problem reported in MySQL#44723. MySQL#44723 is caused by unnecessarily zero-filling the write cache. Users noticed that there is a huge performance difference between inserting 9 rows of data and 11 rows of data. Our statistical debugging points out that the failure is highly related to taking the `(row > MI_MIN_ROWS_TO_USE_WRITE_CACHE)` branch. That is, success runs never take this branch, yet failure runs always take this branch. This is related to the root cause — an inefficient implementation of function `mi_extra`, and the patch makes `mi_extra` more efficient.

Note that identifying the correct failure predictor is not trivial. As shown by the "# of candidate predicates" column of Table 5.5, there is a large number of predicates that have been observed true for at least once in failure runs. Statistical debugging is able to identify the most failure predicting ones out of thousands or even hundreds of thousands of candidate predicates.

Comparing with the profiler For eight cases where the basic statistical model is useful, profilers fail miserably. In terms of identifying root causes (i.e., what causes the inefficient computation), among these 8 cases, the root-cause functions are ranked from number 11 to number 1037 for 5 cases. In the other 3 cases, the function that contains the root cause does not even appear in the profiling result list (i.e., these functions execute for such a short amount of time that they are not even observed by profilers).

Even if we consider functions in the call stacks of top-ranked profiler functions, profiler is helpful for only one out of these eight cases, as shown by the "Stack" column of Table 5.5. That is, for MySQL#44723, the root cause function is the caller's caller of the top ranked function in profiler results. For the other seven benchmarks, the root cause functions do not appear on the call stacks of the top five ranked functions in profile results.

In terms of suggesting fix strategies, profiler results provide no hint about how to solve the performance problem. Instead, the statistical debugging results are informative. For example, among the 7 cases where branch predicates are best failure predictors, the fixes either directly change the branch condition (5 cases) or optimize the code in the body of the branch (2 cases). For the one case where a return predicate is the best failure predictor, the fix affects the return value of the corresponding function.

5.3.2.3 Results for ΔLDA model

We also tried statistical debugging using the ΔLDA model together with the branch, return, and scalar-pair predicates. For branch predicates, we focus on predicates collected at loop-condition branches here and we will refer to them as "$\text{Branch}_{\text{loop}}$" in Table 5.5.

As shown in Table 5.5, ΔLDA model well complements the statistical debugging designs discussed earlier (i.e., basic statistical model). In 11 out of 12 cases where the basic statistical model fails to identify good failure predictors, useful failure predictors are identified by the ΔLDA model.

Among the three different types of predicates, branch predicates are the most useful — help diagnosing 11 cases under ΔLDA model. In general, when a loop-branch predicate b is considered as a failure predictor by the ΔLDA

statistical model, it indicates that b's corresponding loop executes many more iterations during failure runs than during success runs.

In eight cases, the loop ranked number one is exactly the root cause of computation inefficiency. The effect of the patches is to (1) completely remove the inefficient loop from the program; (2) reduce the workload of the loop; or (3) remove redundancy across loop iterations or across loop instances.

In three cases, the root-cause loop is ranked within top four (second, third, and fourth, respectively), but not number one. The reason is that the loop ranked number one is actually part of the *effect* of the performance problem. For example, in GCC#8805 and GCC#46401, the root-cause loop produces more than necessary amount of work for later loops to handle, which causes later loops to execute many more iterations during failure runs than success runs.

In one case, GCC#12322, the root-cause loop is not ranked within top five by ΔLDA model. Similar with GCC#8805 and GCC#46401, the root-cause loop produces many unnecessary tasks. In GCC#12322, these tasks happen to be processed by many follow-up nested loops. The inner loops of those nested loops are all ranked higher than the root-cause loop, as they experience many more iteration-number increases from success runs to failure runs.

Return predicates and scalar-pair predicates can also help diagnose some performance problems under the ΔLDA model, but their diagnosis capability is subsumed by $\text{branch}_{\text{loop}}$ predicates in our evaluation, as shown in Table 5.5. For the six cases when a scalar-pair predicate p is identified as a good failure predictor, p is exactly part of the condition evaluated by a corresponding $\text{branch}_{\text{loop}}$ failure predictor. For the seven cases when a function-return predicate f is identified as a good failure predictor, f is ranked high by the statistical model because it is inside a loop that corresponds to a highly ranked $\text{branch}_{\text{loop}}$ failure predictor.

Comparing with the profiler ΔLDA model is good at identifying root causes located inside loops. Since functions that contain loops tend to rank high by profilers, profilers perform better for this set of performance problems than

	Apache	Chrome	GCC	Mozilla	MySQL	Total
Total # of bugs	16	5	9	19	16	65
# of bugs the default CBI model can help						
Branches	1	0	1	5	5	12
Returns	1	0	0	0	1	2
Scalar-Pairs	0	0	0	0	0	0
# of bugs ΔLDA model can help						
$\text{Branches}_{\text{loop}}$	10	4	8	12	10	44
Returns	0	0	0	0	0	0
Scalar-Pairs	0	0	0	0	0	0
# of bugs above designs cannot help						
	4	1	0	2	0	7

Table 5.6: How different predicates work for diagnosing user-reported performance bugs. In this manual inspection, if more than one predicate can help diagnose a problem, we only count the predicate that is most directly related to the root cause.

the ones discussed in Section 5.3.2.2. In comparison, statistical debugging still behaves better.

In terms of identifying root causes, ΔLDA model always ranks the root-cause loop/function equally good (in 7 cases) or better (in 4 cases) than profilers. There are mainly two reasons that ΔLDA is better. First, sometimes, the root-cause loop does not take much time. They simply produce unnecessary tasks for later loops to process. For example, in GCC#8805, the function that contains the root-cause loop only ranks 20th by profiler. However, it is still ranked high by ΔLDA model, because the loop-iteration-number change is huge between success and failure runs. Second, sometimes, functions called inside an inefficient loop take a lot of time. Profilers rank those functions high, while those functions actually do not have any inefficiency problems.

Considering call-stack functions in the profiling results ("Stack" column in Table 5.5) does not make profiler much more useful. For example, the root cause function of GCC#46401 ranks fifth in the profiling result. This function is also one of the callers' callers of the top-ranked function in the profiling results. However, since the profiler reports three different callers, each having 1–3 callers, for the top-ranked function, the effective ranking for the root-cause function does not change much with or without considering call stacks.

5.3.3 Manual inspection

In addition to the above experimental study, we also manually checked which predicate, if any, would help diagnose each of the 65 user-reported performance bugs in our benchmark set. The result is shown in Table 5.6.

Assuming the basic statistical model, traditional predicates (i.e., branches, returns, and scalar-pairs) can diagnose 14 out of 65 performance problems. Among them, branch predicates are the most helpful, able to diagnose 12 performance problems; return predicates can diagnose 2 performance problems; scalar-pair predicates are the least useful among the three in our study.

Among the ones that cannot be diagnosed by the basic statistical model, 44 of them are caused by inefficient loops. We expect that the ΔLDA statistical model can identify root-cause related branch predicates (denoted as "$\text{Branches}_{\text{loop}}$" in Table 5.6). That is, the loop-condition branch related to the loop that is executed for too many times during failure runs will be ranked high by the ΔLDA model. Some scalar-pair predicates and function-return predicates could also help failure diagnosis under the ΔLDA model. For example, the loop-condition of an inefficient loop could involve the comparison between two scalar variables; the inefficient loop could invoke a function that happens to always return positive values; and so on. However, these predicates will not provide more information than branch predicates. Therefore, we do not mark them in Table 5.6.

The remaining 7 performance problems are mostly caused by unnecessary I/Os or other system calls, not related to any predicates discussed above.

5.3.4 Discussion

Putting our manual inspection results and experimental evaluation results together, we conclude the following:

1. Statistical debugging can help the diagnosis of many user-reported performance problems, improving the state of the art in performance diagnosis;

2. Two design points of statistical debugging are particularly useful for diagnosing performance problems. They are branch predicates under basic

statistical model and branch predicates under ΔLDA model. These two design points complement each other, providing **almost full** coverage of performance problems that we have studied;

3. The basic statistical model that works for most functional bugs [3, 5, 43, 45, 56, 57, 86] is very useful for performance diagnosis too, but still leaves many performance problems uncovered; statistical models that consider the number of times a predicate is true in each run (e.g., the ΔLDA model) is needed for diagnosing performance problems.

4. Statistical debugging alone cannot solve all the problem of diagnosing performance problems. Although statistical debugging can almost always provide useful information for performance diagnosis, developers still need help to figure out the final patches. Especially, when an inefficient loop is pointed out by the ΔLDA model, developers need more program analysis to understand why the loop is inefficient and how to optimize it.

As we discussed in Chapter 3, root causes of performance problems are highly correlated with fixes. For loop-related performance problems, we expect future performance diagnosis systems to use static or dynamic analysis to automatically figure out the fine-grained root causes, differentiate effects from causes, and suggest detailed fix strategies, after statistical debugging identifies root-cause loop candidates.

5.4 Production-run Statistical Debugging

In-house performance diagnosis discussed in Section 5.3 assumes that users file a detailed bug report and developers can repeat the performance problem at the development site. Unfortunately, this does not always happen. In many cases, production-run users only send back a simple automatically generated report, claiming that a failure has happened, together with a small amount of automatically collected runtime information.

The key challenge of diagnosing production-run failures is how to satisfy the following three requirements simultaneously:

1. Low runtime overhead. The diagnosis tool will not be accepted by end users, if it incurs too much slow down to each production run.

2. High diagnosis capability. The diagnosis tool is useful to developers only when it can accurately provide failure root cause information.

3. Short diagnosis latency. Short diagnosis latency can speed up patch design and improve system availability.

This section discusses this issue in the context of performance bugs.

5.4.1 Design

The state of the art production-run functional bug diagnosis [5, 43, 56, 57] proposes to satisfy the first two requirements (i.e., low overhead and high capability) by combining *sampling* techniques with statistical debugging. By randomly sampling predicates at run time, the overhead can be lowered; by processing predicates collected from many failure and success runs together, the diagnosis capability can be maintained for the diagnosis of most functional bugs [5, 43, 56, 57]. The only limitation is that sampling could affect diagnosis latency — the same failure needs to occur for many times until sufficient information can be sampled. This is especially a problem for software that is not widely deployed and bugs that do not manifest frequently. We plan to follow this approach and apply it for production-run performance diagnosis.

Different from production-run functional failure diagnosis [5, 43, 56, 57], production-run performance diagnosis needs to have a slightly different failure-reporting process. Traditional functional failure diagnosis assumes that a profile of sampled predicates will be collected after every run. This profile will be marked as *failure* when software encounters typical failure symptoms such as crashes, error messages, and so on; the profile will be marked as *success* otherwise. The same process does not apply to performance failures, because most performance failures are observed through comparisons across runs, as discussed in Section 5.2.

To adapt to the unique way that performance problems are observed, we expect that users will explicitly mark a profile as *success*, *failure*, or *do-not-care* (the default marking), when they participate in production-run performance

diagnosis. For most performance problems (i.e., those problems observed through comparisons), do-not-care profiles will be ignored during statistical debugging. For performance problems that have non-comparison-based symptoms (i.e. application freeze), all profiles collected from production runs will be considered during statistical debugging.

One issue not considered in this chapter is *failure bucketing*. That is, how to separate failure (or success) profiles related to different software defects. This problem is already handled by some statistical models [56, 57] that can discover multiple failure predictors corresponding to different root causes mixed in one profile pool, as well as some failure bucketing techniques [30] that can roughly cluster profiles based on likely root causes. Of course, performance diagnosis may bring new challenges to these existing techniques. We leave this for future research.

5.4.2 Experimental evaluation

Our evaluation will aim to answer two key questions:

1. Can sampling lower the overhead and maintain the capability of performance-related statistical debugging? A positive answer would indicate a promising approach to production-run performance diagnosis.

2. What is the impact of sampling to diagnosis latency? Traditionally, if we use 1 out of 100 sampling rate, we need hundreds of failure runs to achieve good diagnosis results. Since many performance bugs lead to repeated occurrences of an event at run time, it is possible that fewer failure runs would be sufficient for performance diagnosis. If this heuristic is confirmed, we will have much shorter diagnosis latency than traditional sampling-based failure diagnosis for functional bugs.

5.4.2.1 Methodology

Benchmarks and inputs We reuse the same set of benchmarks shown in Table 5.1. We also use the same methodology to generate inputs and drive success/failure runs. The only difference is that for the four performance

problems that users do not report any good inputs, we will use completely random inputs to produce success-run profiles.

Tool implementation To sample return predicates, we directly use CBI [56, 57]. CBI instruments program source code to conduct sampling. Specifically, CBI instrumentation keeps a global countdown to decide how many predicates can be skipped before next sample. When a predicate is sampled, the global countdown is reset to a new value based on a geometric distribution whose mean value is the inverse of the sampling rate.

To sample branch predicates, we directly use CBI for benchmarks written in C. For all MySQL and some Mozilla benchmarks that are written in C++, since CBI does not work for C++ code, we conduct sampling through hardware performance counters following the methodology described in previous work [5]. Specifically, hardware performance counters are configured so that an interrupt will be triggered every N occurrences of a particular performance event (e.g, branch-taken event), with no changes to the program.

Metrics and settings We will evaluate all three key metrics for failure diagnosis: (1) runtime overhead, measured by the slow down caused by information collection at every run; (2) diagnosis capability, measured by whether top ranked failure predictors are related to failure root causes, as discussed in Section 5.3.2. (3) diagnosis latency, measured by how many failure runs are needed to complete the diagnosis.

By default, we keep the sampling rate at roughly 1 out of 10000 and use samples collected from 1000 failure runs and 1000 success runs for failure diagnosis.

In addition to experiments under the default setting, we also evaluate the impact of different numbers of failure/success runs, ranging from 10 to 1000, while keeping the sampling rate fixed, and evaluate the impact of different sampling rates, ranging from roughly 1 out of 100 to roughly 1 out of 100000, while keeping the number of failure/success runs fixed. Particularly, we will try using only 10 success runs and 10 failure runs, under the default sampling rate, to see if we can achieve good diagnosis capability, low diagnosis latency, and low runtime overhead *simultaneously*.

Since sampling is random, we have repeated our evaluation for several rounds to confirm that all the presented results are stable.

For every performance problem benchmark, the results presented below are obtained under the combination of predicate and statistical model that is shown to be (most) effective in Table 5.5 (Section 5.3). That is, basic model plus branch predicates are used for seven benchmarks; basic model plus return predicates are used for one benchmark; ΔLDA model plus $branch_{loop}$ predicates are used for the remaining twelve benchmarks, including GCC#12322. Since sampling can only lower overhead and cannot improve the diagnosis capability, those combinations that fail to deliver useful diagnosis results in Table 5.5 still fail to deliver useful diagnosis results in our sampling-based evaluation.

5.4.2.2 Results

BugID	Diagnosis Capability				Overhead
(# of runs)	(10)	(100)	(500)	(1000)	per run
Mozilla258793	-	✓$_1$	✓$_1$	✓$_1$	2.39%
Mozilla299742	-	-	✓$_1$	✓$_1$	4.27%
Mozilla347306	✓$_1$	✓$_1$	✓$_1$	✓$_1$	1.42%
Mozilla416628	✓$_1$	✓$_1$	✓$_1$	✓$_1$	2.03%
MySQL15811	✓$_1$	✓$_1$	✓$_1$	✓$_1$	2.25%
MySQL26527	-	-	✓$_1$	✓$_1$	6.05%
MySQL27287	✓$_1$	✓$_1$	✓$_1$	✓$_1$	3.02%
MySQL40337	-	✓$_1$	✓$_1$	✓$_1$	2.69%
MySQL42649	-	-	✓$_2$	✓$_1$	6.10%
MySQL44723	-	✓$_1$	✓$_1$	✓$_1$	3.16%
Apache3278	-	-	-	-	0.23%
Apache34464	✓$_3$	✓$_3$	✓$_3$	✓$_3$	0.18%
Apache47223	✓$_1$	✓$_1$	✓$_1$	✓$_1$	0.13%
Apache32546	✓$_1$	✓$_1$	✓$_1$	✓$_1$	0.38%
GCC1687	✓$_1$	✓$_1$	✓$_1$	✓$_1$	0.80%
GCC8805	✓$_4$	✓$_4$	✓$_4$	✓$_4$	1.81%
GCC15209	-	-	✓$_1$	✓$_1$	2.37%
GCC21430	✓$_1$	✓$_1$	✓$_1$	✓$_1$	7.55%
GCC46401	✓$_2$	✓$_2$	✓$_2$	✓$_2$	2.91%
GCC12322	-	-	-	-	2.33%

Table 5.7: Runtime overhead and diagnosis capability evaluated with the default sampling rate (1 out of 10000); 10, 100, 500, 1000 represents the different numbers of success/failure runs used for diagnosis.

BugID	Diagnosis Capability				Overhead				Avg. # of sampled predicates			
(sampling rate)	($\frac{1}{10^2}$)	($\frac{1}{10^3}$)	($\frac{1}{10^4}$)	($\frac{1}{10^5}$)	($\frac{1}{10^2}$)	($\frac{1}{10^3}$)	($\frac{1}{10^4}$)	($\frac{1}{10^5}$)	($\frac{1}{10^2}$)	($\frac{1}{10^3}$)	($\frac{1}{10^4}$)	($\frac{1}{10^5}$)
Mozilla258793	*	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	*	24.36%	2.39%	1.84%	*	$1.42*10^6$	$1.45*10^5$	$1.49*10^4$
Mozilla299742	*	$\checkmark_1$	$\checkmark_1$	$\checkmark_2$	*	30.84%	4.27%	4.16%	*	$1.87*10^5$	$1.77*10^4$	$1.82*10^3$
Mozilla347306	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	69.73%	8.27%	1.42%	0.56%	$7.13*10^6$	$7.13*10^5$	$7.13*10^4$	$7.13*10^3$
Mozilla411722	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	24.64%	4.31%	2.03%	1.36%	$8.18*10^5$	$8.18*10^4$	$8.17*10^3$	816.56
MySQL15811	*	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	*	7.65%	2.25%	1.53%	*	$3.67*10^5$	$1.67*10^5$	$1.66*10^4$
MySQL26527	*	$\checkmark_1$	$\checkmark_1$	-	*	6.40%	6.05%	4.53%	*	$3.23*10^3$	921.41	92.60
MySQL27287	*	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	*	4.63%	3.02%	0.61%	*	$2.52*10^6$	$1.15*10^6$	$1.19*10^5$
MySQL40337	*	$\checkmark_1$	$\checkmark_1$	-	*	10.88%	2.69%	2.28%	*	$5.10*10^6$	$1.66*10^6$	$1.42*10^5$
MySQL42649	*	$\checkmark_1$	$\checkmark_1$	-	*	8.28%	6.10%	3.93%	*	$7.25*10^3$	$1.14*10^3$	128.53
MySQL44723	*	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	*	7.10%	3.16%	2.24%	*	$3.23*10^5$	$1.83*10^5$	$1.46*10^4$
Apache3278	$\checkmark_1$	-	-	-	0.23%	0.23%	0.23%	0.23%	0.21	0.01	0	0
Apache34464	$\checkmark_3$	$\checkmark_3$	$\checkmark_3$	$\checkmark_3$	29.45%	2.62%	0.18%	0.04%	$2.50*10^7$	$2.50*10^6$	$2.49*10^5$	$2.50*10^4$
Apache47223	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	12.58%	1.28%	0.13%	0.12%	$6.27*10^6$	$6.26*10^5$	$6.27*10^4$	$6.27*10^3$
Apache32546	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	0.24%	0.39%	0.38%	0.40%	$9.75*10^3$	977.72	99.01	9.5
GCC1687	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	47.30%	5.34%	0.80%	0.43%	$3.18*10^7$	$3.18*10^6$	$3.18*10^5$	$3.17*10^4$
GCC8805	$\checkmark_4$	$\checkmark_4$	$\checkmark_4$	$\checkmark_4$	50.92%	7.33%	1.81%	1.05%	$1.63*10^7$	$1.63*10^6$	$1.63*10^5$	$1.63*10^4$
GCC15209	$\checkmark_1$	$\checkmark_2$	$\checkmark_1$	$\checkmark_2$	41.06%	8.43%	2.37%	1.27%	$3.35*10^4$	$3.35*10^3$	334.72	33.64
GCC21430	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	64.98%	13.68%	7.55%	5.07%	$9.15*10^7$	$9.15*10^6$	$9.15*10^5$	$9.15*10^4$
GCC46401	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	$\checkmark_2$	88.97%	13.04%	2.91%	0.46%	$8.88*10^7$	$8.88*10^6$	$8.88*10^5$	$8.88*10^4$
GCC12322	-	-	-	-	15.55%	2.33%	2.33%	0.56%	$9.97*10^7$	$9.97*10^6$	$9.97*10^5$	$9.97*10^4$

Table 5.8: Diagnosis capability, overhead, and average number of samples in each run under different sampling rates by using 1000 success/failure runs. *: no results are available, because hardware-based sampling cannot be as frequent as 1/100 and software-based CBI sampling does not apply for these C++ benchmarks.

Runtime overhead As shown in Table 5.7, the runtime overhead is small under the default sampling rate (i.e., 1 out of 10000). It is below 5% in all but three cases, and is always below 8%.

As expected, the overhead is sensitive with the sampling rate. As shown in Table 5.8, it can be further lowered to be mostly below 2% under the $\frac{1}{10^5}$ sampling rate, and could be as large as over 40% under the $\frac{1}{100}$ sampling rate.

Diagnosis capability As shown by Table 5.7, with 1000 success runs and 1000 failure runs, sampling did very little damage to the diagnosis capability of statistical debugging. Apache#3278 is the only one, among all benchmarks, where failure diagnosis fails under this sampling setting. For all other bench-

marks, the rankings of the ideal failure predictors remain the same as those without sampling in Table 5.5.

Also as expected, the diagnosis capability would decrease under sparser sampling or fewer failure/success runs. As shown in Table 5.8, under the default setting of 1000 success/failure runs, the diagnosis capability is roughly the same between $\frac{1}{10^3}$ sampling rate and $\frac{1}{10^4}$ sampling rate, but would drop with $\frac{1}{10^5}$ sampling rate. Four benchmarks that can be diagnosed with more frequent sampling cannot be diagnosed with $\frac{1}{10^5}$ sampling rate. Clearly, more runs will be needed to restore the diagnosis capability with a lower sampling rate.

Diagnosis latency Diagnosis latency versus runtime overhead and diagnosis capability is a fundamental trade-off facing sampling-based statistical debugging for functional bugs [43, 56, 57]. With sampling, intuitively, more failure runs are needed to collect sufficient diagnosis information. This is **not** a problem for widely deployed software projects. In those projects, the same failure tends to quickly occur for many times on many users' machines [30]. However, this is a problem for software that is not widely deployed.

We quantitatively measured the impact of sampling to diagnosis latency in Table 5.7. As we can see, three benchmarks need about 100 failure runs for their sampling-based diagnosis to produce useful results; four benchmarks need about 500 failure runs; and one benchmark, Apache#3278, needs more than 1000 failure runs. This indicates longer diagnosis latencies than the non-sampling-based diagnosis evaluated in Section 5.3, where only 10 failure runs are used.

Interestingly, there are 11 benchmarks, whose diagnosis latency is **not** lengthened by sampling. As shown in Table 5.7, even with only 10 failure runs, the sampling-based diagnosis still produces good failure predictors. These are exactly all the 11 benchmarks that ΔLDA model suits in Table 5.5. For all these benchmarks, the rankings are exactly the same with or without sampling, with just 10 failure runs. Consequently, sampling allows us to achieve low runtime overhead (<10%), high diagnosis capability, and low diagnosis latency

simultaneously, a feat that is almost impossible for sampling based functional bug diagnosis.

The nice results for these 11 benchmarks can be explained by a unique feature of performance bugs, especially loop-related performance bugs — their root-cause related predicates are often evaluated to be true for many times in one run, which is why the performance is poor. Consequently, even under sparse sampling, there is still a high chance that the root-cause related predicates can be sampled, and be sampled more frequently than root-cause unrelated predicates.

Finally, even for the other 9 benchmarks, $\frac{1}{10^4}$ sampling rate does not extend diagnosis latency by 10^4 times. In fact, for most of these benchmarks, 100 – 500 failure runs are sufficient for failure diagnosis under $\frac{1}{10^4}$ sampling rate. Our investigation shows that the root-cause code regions in these benchmarks are all executed for several times during the user-reported failure runs, which is likely part of the reason why users perceived the performance problems. Consequently, the negative impact of sampling on diagnosis latency is alleviated.

5.5 Conclusion

Software design and implementation defects lead to not only functional misbehavior but also performance losses. Diagnosing performance problems caused by software defects are both important and challenging. This chapter made several contributions to improving the state of the art of diagnosing real-world performance problems. Our empirical study showed that end users often use comparison-based methods to observe and report performance problems, making statistical debugging a promising choice for performance diagnosis. Our investigation of different design points of statistical debugging shows that branch predicates, with the help of two types of statistical models, are especially helpful for performance diagnosis. It points out useful failure predictors for 19 out of 20 real-world performance problems. Furthermore, our investigation shows that statistical debugging can also work for production-run performance diagnosis with sampling support, incurring less than 10% overhead in our evaluation. Our study also points out directions for future work on fine-granularity performance diagnosis.

Chapter 6

LDoctor: Hybrid Analysis Routines for Inefficient Loops

Statistical performance debugging discussed in the last chapter can identify which loop is the root cause for a performance problem. However, it cannot provide detailed root-cause information, such as why the loop is inefficient and how developers might fix the problem.

In this chapter, we first conduct an empirical study to understand what is fine-grained root-cause information for inefficient loops in the real world. We then design a series of static-dynamic hybrid analysis routines that can help identify accurate fine-grained root-cause information. We further use sampling techniques to lower our diagnosis overhead without hurting diagnosis accuracy or latency. Evaluation using real-world inefficient loops shows that our tool can provide good coverage and accuracy, with small runtime overhead.

6.1 Introduction

6.1.1 Motivation

In Chapter 5, we discussed how to apply statistical debugging to performance failure diagnosis. Given the symptom of a performance problem (e.g., two similar inputs leading to very different execution speed), statistical debugging, an approach widely used for diagnosing functional failures [45, 56, 57], can accurately identify control-flow constructs that are most correlated with the performance problem by comparing problematic runs and regular runs.

Unfortunately, for loop-related performance problems, which contribute to two-thirds of user-reported real-world performance problems studied in Chapter 5, statistical debugging is not very effective. Although it can identify the root-cause loop, it does not provide any information regarding why the loop is inefficient and hence is not very helpful in fixing the performance problem.

Figure 1.1 shows a real-world performance bug in GCC. Function `synth_mult` is used to compute the best algorithm for multiplying `t`. It is so time consuming that developers tried speeding it up through memoization — hash-table `alg_hash` is used to remember which `t` has been processed in the past and what is the processing result. Unfortunately, a small mistake in the type declaration of hash-table entry `alg_hash_entry` causes memoization not to work when `t` is larger than the maximum value type-`int` can represent. As a result, under certain workloads, `synth_mult` conducts a lot of redundant computation for the same `t` values recursively, leading to huge performance problems. In practice, developers know the problem is inside `synth_mult` very early. However, since this function is complicated, it took them several weeks to figure out the root cause. If a tool can tell them not only which loop[1] is the root cause but also why the loop is inefficient (i.e., lot of redundant computation and the need for better memoization in this example), the diagnosis and bug fixing process would be much easier for developers.

Clearly, more research is needed to improve the state of the art of performance diagnosis. Diagnosis techniques that can provide fine-grained root-cause information for common performance problems, especially loop-related performance problems, are well desired.

6.1.2 Contributions

This chapter presents a tool, LDoctor, that can effectively provide fine-grained root causes for inefficient loops.

LDoctor targets the most common type of performance problems, inefficient loops, according to empirical studies in Chapter 3 and Chapter 5. LDoctor also targets a challenging problem. Although statistical debugging (Chapter 5) can help identify suspicious loops that are highly correlated with user-

[1] Recursive functions are handled similarly as loops in this chapter.

perceived performance problems, it provides no information about why the suspicious loop is inefficient, not to mention providing fix suggestions. Although various bug-detection techniques [72, 73, 75] can help detect loop-related performance bugs, they all suffer from generality problems, covering only a specific type of loop problems each; the dynamic tools [72, 75] also suffer from performance problem, imposing 10X slowdown or more.

LDoctor can provide good coverage, accuracy and performance simultaneously: 1) Coverage. LDoctor can cover most common root causes for inefficient loops. 2) Accuracy. LDoctor can accurately point out whether a loop is inefficient, explain why the loop is inefficient, and provide correct fix suggestions. 3) Performance. LDoctor incurs a low runtime overhead.

We build LDoctor through the following three steps.

First, figuring out a taxonomy for the root causes of common inefficient loops. Such a taxonomy is the prerequisite to developing a general and accurate diagnosis tool. Guided by a thorough study of 45 real-world inefficient loop problems, we come up with a hierarchical root-cause taxonomy for inefficient loops. Our empirical study shows that our taxonomy is general enough to cover common inefficient loops, and also specific enough to help developers understand and fix the performance problem. We will present more details in Section 6.2.

Second, building a tool(kit) LDoctor that can automatically and accurately identify the fine-grained root-cause type of a suspicious loop and provide fix-strategy suggestions. We achieve this following several principles:

- Focused checking. Different from performance-bug detection tools that blindly check the whole software, LDoctor focuses on performance symptoms and only check inefficient loop candidates identified by existing tools, a statistical debugging tool (Chapter 5) in our current prototype. This focused study is crucial for LDoctor to achieve high accuracy.

- Root-cause taxonomy guided design. To provide the needed coverage, we follow the root-cause taxonomy discussed above and design analysis routines for every root-cause sub-category. Given a candidate inefficient

loop, we will apply a series of analysis to it to see if it matches any type of inefficiency.

- Static-dynamic hybrid analysis to balance performance and accuracy. As we will see, static analysis alone cannot accurately identify inefficiency root causes, especially because some inefficiency problems only happen under specific workload. However, pure dynamic analysis will cause too large runtime overhead. Therefore, we take a hybrid approach to achieve both performance and accuracy goals.

Third, using sampling to further lower the runtime overhead of LDoctor, without degrading diagnosis capability. Random sampling is a natural fit for performance diagnosis due to the repetitive nature of inefficient codes, especially inefficient loops.

We have evaluated LDoctor on 18 real-world performance problems. Evaluation results show that LDoctor can accurately identify the detailed root cause for all benchmarks and provide correct fix-strategy suggestion for 16 benchmarks. All these are achieved with low runtime overhead.

6.2 Root-Cause Taxonomy

The first task we are facing is to figure out a root-cause taxonomy for real-world inefficient loops. Our taxonomy design is guided by checking inefficient loops in the benchmark suite discussed in Chapter 3 and Chapter 5. We also try to satisfy three requirements in our design. (1) Coverage: covering a big portion of real-world inefficient loop problems; (2) Actionability: each root-cause category should be informative enough to help developers decide how to fix a performance problem; (3) Generality: too application-specific root causes will not work, as we need to build diagnosis tools to automatically identify these root causes without developers' help. In this section, we will first present our taxonomy, together with real-world examples, followed by our empirical study about how a suite of real-world inefficient loop problems fall into our root-cause categories.

```
//Mozilla347306
//jsscript.c

jssrcnote *
js_GetSrcNote(JSScript *script, jsbytecode *pc)
{
    ...
    target = PTRDIFF(pc, script->code, jsbytecode);
    for (sn = SCRIPT_NOTES(script);
         !SN_IS_TERMINATOR(sn); sn = SN_NEXT(sn))
    {
        offset += SN_DELTA(sn);
        if (offset == target && SN_IS_GETTABLE(sn))
            return sn;
    }
    return NULL;
}
```

Figure 6.1: A resultless 0*1? bug in Mozilla

6.2.1 Taxonomy

Our taxonomy contains two major root cause categories, *resultless* and *redundancy*, and several sub-categories, as below.

Resultless Resultless loops spend a lot of time in computation that does not produce any results that will be used after the loop. Depending on when such resultless computation is conducted, we further consider four sub-categories.

0*: This type of loops never produce any results in any iteration. This type of loops should be rare in mature software systems. They should simply be deleted from the program.

0*1?: This type of loops only produce results in the last iteration, if any. They are often related to search: they check a sequence of elements one by one until the right one is found. Not all loops of this type are inefficient. If they are, they are often fixed by data-structure changes [2]. Figure 6.1 shows an example from Mozilla JavaScript engine. The loop searches through an array containing source code information for the input `pc`. In practice, this loop needs to execute more than 10000 iterations for many long JavaScript encountered by users, which leads to huge performance problems. To fix this

[2] All bugs fixed by data-structure change are categorized as *In-place Call Change* in Chapter 3.

```
//GCC46401
//c−common.c

//bad input contains a long expression:
//QStringList function_list =
//                QStringList() << "abasep" <<"abs" ...

static bool
candidate_equal_p (const_tree x, const_tree y)
{
   return (x == y) || (x && y && operand_equal_p (x, y, 0));
}

//comments from developers:
//"No point tracking CALL_EXPRs that aren't ECF_CONST
//(because  then operand_equal_p fails anyway) nor
//STRING_CSTs (which can't be written into)"
static void
merge_tlist(struct tlist **to, struct tlist *add, int copy)
{
   ...
   for (tmp2 = *to; tmp2; tmp2 = tmp2->next)
      if (candidate_equal_p (tmp2->expr, add->expr))
      {
         found = 1;
         if (!tmp2->writer)
            tmp2->writer = add->writer;
      }
   ...
}
```

Figure 6.2: A resultless [0|1]* bug in GCC

problem, developers simply replace array with hash table when processing long JavaScript files.

[0|1]*: Every iteration in this type of loops may or may not produce results. Under certain workload, the majority of the loop iterations do not produce any results and lead to performance problems perceived by end users. The inefficiency problem caused by this type of loops can often be solved by adding extra conditions to skip the loop under certain contexts. Figure 6.2 shows such an example from GCC. Only when the if branch executed, one iteration will generate results. The bug happens when GCC checks violations of sequence

```
1  //Mozilla477564
2  //nsSessionStore.js
3
4  generate: function sss_xph_generate(aNode)
5  {
6      ...
7      for (let n = aNode; (n = n.previousSibling); )
8          if (n.localName == aNode.localName && n.namespaceURI ==
9            aNode.namespaceURI && (!nName || n.name == nName))
10             count++;
11     //count will be used to generate a ID
12     //for the aNode
13     ...
14 }
```

Figure 6.3: A cross-loop redundant bug in Mozilla

point rule. The user notices severe performance degradation when enabling the checking. The buggy loop is super-linear inefficient in terms of the number of operands in one expression. The buggy input contains an expression, which is long and special. The loop takes a lot of iterations during bad run, but none of the iterations will generate results. The patch is to add extra checking before processing each operand. When it also has the special feature like the bad input, the loop will be skipped.

[1]*: Loops in this category always generate results in almost all iterations. They are inefficient, because the results generated could be useless due to some high-level semantic reasons. Understanding and fixing this type of inefficiency problems often require deep understanding of the program and are difficult to automate. For example, several Mozilla performance problems are caused by loops that contain intensive GUI operations. Although every iteration of these loops produce side effects, the performance problem forced developers to change the program and batch/skip some GUI operations.

Redundant Redundant loops spend a lot of time in repeating computation that is already conducted. Depending on the unit of the redundant computation, we further consider two sub-categories.

Cross-iteration Redundancy: Loop iteration is the redundancy unit here: one iteration repeats repeating what was already done by an earlier iteration

of the same loop. Here, we consider recursive function calls as a loop, treating one function call instance as one loop iteration. This type of inefficiency is often fixed by memoization or batching, depending on whether the redundancy involves I/O operations. For example, GCC#27733 shown in Figure 1.1 is fixed by better memoization. Mozilla#490742 in Figure 3.4 represents a slightly different type of cross-iteration redundancy. The inefficient loop in this case saves one URL into the "Favorite Links" database in each iteration. One database transaction in each iteration turns out to be to time consuming, with too much redundant work across iterations. At the end, developers decide to batch all database updates into one big transaction, which speeds up some workload like bookmarking 50 tabs from popping up timeout windows to not blocking.

Cross-loop Redundancy: A whole loop is the redundancy unit here: one dynamic instance of a loop spends a big chunk, if not all, of its computation in repeating the work already done by an earlier instance of the same loop. Developers often fix this type of inefficiency problems through memoization: caching the earlier computation results and skip following redundant loops. Mozilla#477564 shown in Figure 6.3 is an example for this type of bugs. The buggy loop is to count how many previous siblings of the input `aNode` have the same name and URI. There is an outer loop for the buggy one, and the outer loop will update `aNode` by using its next siblings. The bug is fixed by saving the calculated count for each node. A new count value is calculated by adding one to the saved count value of the nearest previous sibling with the same name and URI.

6.2.2 Empirical study

Having presented our taxonomy above, we will see how it works for a set of real-world inefficient loop problems collected in Chapter 3 and Chapter 5. Specifically, we want to check the coverage and the actionability of our taxonomy presented above: (1) How common are real-world performance problems caused by the patterns discussed above? (2) How are real-world problems fixed by developers? Can we predict their fix strategies based on the root-cause pattern?

6.2.2.1 Methodology

Application Suite Description (language)	# Bugs
Apache Suite HTTPD: Web Server (C) TomCat: Web Application Server (Java) Ant: Build management utility (Java)	11
Chromium Suite Google Chrome browser (C/C++)	4
GCC Suite GCC & G++ Compiler (C/C++)	8
Mozilla Suite Firefox: Web Browser (C++, JavaScript) Thunderbird: Email Client (C++, JavaScript)	12
MySQL Suite Server: Database Server (C/C++) Connector: DB Client Libraries (C/C++/Java/.Net)	10
Total	45

Table 6.1: Applications and bugs used in the study.

In Chapter 3, we studied the on-line bug databases of five representative open-source software projects, as shown in Table 6.1. Through a mix of random sampling and manual inspection, we found 65 performance problems that are perceived and reported by users in Chapter 5. Among these 65 problems, 45 problems are related to inefficient loops and hence are the target of the study here[3].

6.2.2.2 Observations

Are the root-causes in our taxonomy common? The answer is *yes*. Resultless loops are about as common as redundant loops (24 vs. 21). Intuitively, 0* loops, where no iteration produces any results, are rare in mature software. In fact, no bugs in this benchmark suite belong to this category. All other root-cause sub-categories are well represented.

How are real-world performance bugs fixed? Overall, the fix strategies are well aligned with root-cause patterns. For example, all inefficient loops with resultless [0|1]* are fixed by skipping the loop under certain contexts [4], and all

[3]The definition of "loop-related" in this chapter is a little bit broader than in Chapter 5, which only considers 43 problems as loop-related.

[4]All bugs fixed by skipping the loop are categorized as *Change Condition* in Chapter 3.

	Apache	Chrome	GCC	Mozilla	MySQL	Total	Fix Strategy
Total # of loop-related bugs	11	4	8	12	10	45	
			# of *Resultless* bugs				
0*	0	0	0	0	0	0	
0*1?	0	0	0	2	3	5	C(4)|S(1)
[0|1]*	0	1	1	1	1	4	S(4)
1*	1	2	3	6	3	15	B(4)|S(4)|O(7)
			# of *Redundant* bugs				
Cross-**iteration** redundancy	7	1	2	1	1	12	B(4)|M(8)
Cross-**loop** redundancy	3	0	2	2	2	9	B(4)|M(5)

Table 6.2: Number of bugs in each root-cause category. B, M, S, C, and O represent different fix strategies: B(atching), M(emoization), S(kipping the loop), C(hange the data structure), and O(thers). The numbers in the parentheses denote the number of problems that are fixed using specific fix strategies.

inefficient loops with redundant root causes are fixed either by memoization or batching the computation. In fact, as we will discuss later, simple analysis can tell whether memoization or batching should be used to fix a problem. The only problem is that there are no silver bullets for 1* loops.

Implications As we can see, resultless loops and redundant loops are both common reasons that cause user-perceived performance problems in practice. The root-cause categories discussed above also match with developers' fix strategy well — with a little bit amount of extra analysis, one can pretty much predict the fix strategy based on the root-cause pattern. Consequently, tools that cover these root causes will have high chances to satisfy the coverage and accuracy requirements of performance diagnosis.

6.2.2.3 Caveats

Just as previous empirical study work, our empirical study above needs to be interpreted with our methodology in mind. As we discussed in Chapter 3 and Chapter 5, we use developers tagging and on-line developer/user discussion to judge whether a bug report is about performance problems and whether the performance problem under discussion is noticed and reported by users or not. We follow the methodology used in Chapter 5 to judge whether the root cause of a performance problem is related to loops or not. We do not

intentionally ignore any aspect of loop-related performance problems. Some loop-related performance problems may never be noticed by end users or never be fixed by developers, and hence skipped by our study. However, there are no conceivable ways to study them.

We believe that the bugs in our study provide a representative sample of the well-documented and fixed performance bugs that are user-perceived and loop-related in the studied applications. Since we did not set up the root-cause taxonomy to fit particular bugs in this bug benchmark suite, we believe our taxonomy and diagnosis framework presented below will go beyond these sampled performance bugs.

6.3 LDoctor Design

LDoctor is composed of a series of analysis routines, each designed to identify whether a given loop belongs to one specific type of root causes, as we will present below.

As briefly mentioned in Section 6.1, the design of these analysis routines follows the following principles.

- Providing diagnosis information, not detecting bugs. LDoctor will be used together with other performance diagnosis tools (Chapter 5) and focus on a small number of loops that are most correlated with a specific performance symptom, instead of being applied to the whole program. Therefore, we will have different design trade-offs in terms of coverage and accuracy, comparing with bug detection tools.

- Static-dynamic hybrid analysis. As we will see, static analysis alone will not be able to provide all the needed information for performance diagnosis; dynamic analysis alone will incur too much unnecessary overhead. Therefore, we will use static-dynamic hybrid approach throughout our design.

- Using sampling to decrease runtime overhead. Loop-related performance problems have the unique nature of repetitiveness, which make them a natural fit for random sampling. Therefore, we will design different sampling schemes for different analysis routines.

6.3.1 Resultless checker

Our resultless checker includes two parts. First, we use static analysis to figure out which are the side-effect instructions in a loop and hence decide whether a loop belongs to 0*, 0*1?, [0|1]*, or 1*. Second, for 0*1? and [0|1]* loops, we use dynamic analysis to figure out what portion of loop iterations are resultless at run time, which will help decide whether the loop is indeed inefficient.

6.3.1.1 Static analysis

We consider side-effect instructions as those instructions that write to variables defined outside the loop. The analysis to identify side-effect instructions is straightforward. We consider all functions that are called by a loop directly or indirectly — a function F that updates variables defined outside F makes the corresponding call statement in F's caller a side-effect instruction. We consider all library functions or function calls through function pointers as functions that have side effects, unless the library functions are specially marked by us in a white list.

After identifying side-effect instructions, it is straight-forward to categorize loops into the four types discussed above. Loop 0* contains no side-effect instructions. Loop 1* contains at least one side-effect instruction along every path that starts from the loop header and ends at the loop header. The remaining cases are either 0*1? or [0|1]*. Differentiating these two cases is also straight-forward. In short, when the basic block that contains side-effect instructions is part of the natural loop, the case belongs to [0|1]*; instead, if the side-effect basic block is strictly post-dominated by one of the loop-exit blocks and is dominated by the loop header, yet is not part of the natural loop, the case belongs to 0*1?.

Finally, since the 1* pattern contains the least amount of information about computation *inefficiency*, LDoctor will not report a loop's root-cause type as 1*, if more informative root-cause type is identified for this loop (e.g., cross-iteration or cross-loop redundancy).

6.3.1.2 Dynamic monitoring

Except for 0*, none of the other three type of loops are inefficient for sure. We need dynamic analysis to figure out what portion of loop iterations are resultless at run time, which will help decide whether the loop is indeed the root cause of a user-perceived performance problem and worth fixing.

For a 0*1? loop, since it only generates results in the last iteration, we only need to know the total number of iterations (or the average total number of iterations when the loop has multiple instances) to figure out the *resultless rate* of the loop. The implementation is straightforward — we initialize a local counter to be 0 in the pre-header of the loop; we increase the counter by 1 in the loop header to count the number of iterations; we dump that local counter value to a global counter when the loop exits.

For [0|1]*, we need to count not only the total number of iterations, but also the exact number of iterations that execute side-effect instructions at run time. To do that, our instrumentation uses a local boolean variable `HasResult` to represent whether one iteration have side effect or not. `HasResult` is set to `False` in the loop header, and set to `True` after each side-effect instruction. It will be used to help count the number of side-effect iterations. For performance concerns, before instrumenting side-effect blocks, we check whether there are post-domination relation between each pair of side-effect blocks. If both block A and block B are side-effect blocks and block A post-dominates block B, we only instrument block A to update `HasResult`.

We could speed up the above counting using sampling. However, since the runtime overhead of the above counting is low, as shown in Section 6.4, our current prototype of LDoctor does not use sampling for this part of runtime analysis.

6.3.1.3 Limitations

The technique designed in this section has the following limitations. First, when callee may have side effect, we will consider it will have side effect in the caller side, and do not consider the real execution inside callee. This could bring false negatives, because we could miss resultless cases, where side effect

instructions inside callee do not execute. Experiments results in Section 6.4 show that this is not a big issue, since we do not miss any resultless bugs.

Second, our dynamic instrumentation does not consider concurrent execution of the monitored loop, because most of buggy loops we study only execute in one single thread. When the monitored loop is executed in multi-thread, like loop marked with omp pragma, we need to synchronize updates to global variables.

6.3.2 Redundancy checker

6.3.2.1 Design overview

To check whether there is redundant computation across different iterations of one loop instance or across different instances of one static loop, we need to address several challenges.

How to judge redundancy between two iterations/loop-instances? Given two iterations (or loop-instances) i_1 and i_2, since they have the same source code, a naive, yet expensive, solution is to record and compare the return value of every memory read conducted by i_1 and i_2. Two better alternative solutions are to record and compare only the values written or read by the side-effect instructions, such as line 10 in Figure 6.3, or the source instructions[5], such as source[i] at line 11 of Figure 6.4.

Among the the above two potential solutions, our design chooses the second one. The reason is that, if there is indeed redundancy, repetitive patterns at the side-effect instructions are caused by repetitive patterns in the source instructions. For the purpose of performance diagnosis, it is more informative to track the cause, rather than the effect.

How to handle partial redundancy? In practice, redundant loops may be doing largely the same, instead of exactly the same, computation across iterations or loop instances. It is also possible that only some, instead of all, iterations in a loop are doing redundant computation. We will discuss how we handle this issue in Section 6.3.2.3.

[5]We define source instructions in a code region r as a set of memory-read instructions that side-effect instructions in r depend on and do not depend on any other instructions inside r.

```
1  //Apache34463
2  //java.lang.String
3  static int indexOf(char[] source, int sourceCount, char[] target, int targetCount )
4  {
5      ...
6      char first  = target[targetOffset];
7      int max = sourceCount − targetCount;
8      for (int i = 0; i <= max; i++) {
9        // Look for first character.
10       if (source[i] != first) {
11           while (++i <= max && source[i] != first);
12       }
13       // Found first character now look at the rest
14       if (i <= max) {
15           int j = i + 1;
16           int end = j + targetCount − 1;
17           for (int k = 1; j < end && source[j] == target[k]; j++, k++);
18           if (j == end) {
19               /* Found whole string. */
20               return i ;
21           }
22       }
23     }
24     return −1;
25 }
26 //TelnetTask.java
27 public void waitForString(...)
28 {
29 +     int windowIndex = −s.length();
30 −     while (sb.toString().indexOf(s) == −1) {
31 +     while (windowIndex++ < 0 || sb.substring(windowIndex).indexOf(s) == −1) {
32         sb.append((char) is.read());
33     }
34 }
```

Figure 6.4: A cross-loop redundant bug in Apache

How to lower the overhead of record-and-compare? Even if we only record and compare the values returned by source instructions, instead of all instructions in a loop, the runtime overhead and the log size would still be large. We will use two ways to lower this time and spatial overhead. First, static analysis can already tell some source instructions will always return the same value across iterations/loop-instances, and hence need not be traced at run time. Second, we can leverage the repetitive nature of performance problems and use random sampling to lower the overhead without degrading the diagnosis capability. We will discuss details of these two optimization in Section 6.3.2.4 and 6.3.2.5.

How to provide the most suitable fix-strategy suggestion? Finally, as discussed in Section 6.2.2.2 and Table 6.2, memoization and batching are both common fix strategies for redundant loops. To pick the right fix strategies to fix a redundant loop, we will conduct some extra analysis. We will discuss this in Section 6.3.2.6.

6.3.2.2 Identifying source instructions

Informally, we use static analysis to identify a set of memory-read instructions that the loop computation depends on. We refer to these instructions as *source* instructions. The values returned from them at runtime will be tracked and compared to identify redundant computation.

Specifically, we first identify side-effect instructions in the loop, as discussed in Section 6.3.1.1; we then conduct static slicing from these instructions, considering both control and data dependency, to identify source instructions.

Our slicing ends when it reaches either a local-variable read conducted outside the loop or a heap/global-variable read anywhere in the program. For the latter case, our slicing stops because tracking data-dependency through heap/global variables is complicated in multi-threaded C/C++ programs. For the former case, not including local-variable reads inside the loop can help reduce the amount of data that needs to be recorded. When there are function calls inside the loop, we conduct slicing for return values of callees and side-effect instructions inside callees. We omit encountered constant

values through slicing, because constant values will not influence whether a loop or an iteration is redundant.

The analysis for cross-iteration and cross-loop redundancy analysis is pretty much the same. The only difference is that, if a memory read instruction `i` in a loop depends on the value returned by instruction `j` in an earlier iteration, we stop tracing the dependence at `i` and consider `i` as a source instruction for cross-iteration redundancy analysis, while we continue the slicing for cross-loop redundancy analysis. For example, the instruction defining the value of `i` is the only side-effect instruction for the loop at line 11 of Figure 6.4. The source instructions calculated by cross-loop dependence analysis include memory read `source[i]` inside the loop, and three values defined outside the loop, which represent the initial value of `i`, the value of `max` and `first` respectively. In contrast, the source instructions calculated by cross-iteration dependence analysis include value `i` defined in previous iteration, memory read `source[i]`, and two values defined outside the loop (`max` and `first`).

6.3.2.3 Identifying redundant loops

After identifying source instructions, we instrument these source instructions, so that the values returned by these memory read instructions can be recorded at run time. Specifically, we will assign a unique ID for each source instruction, and a pair of $< \mathtt{InstID}, \mathit{Value} >$ will be recorded at run time with the execution of a source instruction. Our trace also includes some delimiters and meta information that allows trace analysis to differentiate values recorded from different loop iterations, different loop instances, and so on.

After collecting values returned by source instructions from every iteration of one or multiple loop instances, we need to process the trace and decide whether the loops under study contain cross-iteration redundancy or cross-loop redundancy. We will first present our high-level algorithms, followed by the exact implementation in our prototype.

High-level algorithms For cross-iteration redundancy, we need to answer two questions. First, how to judge whether two iterations are doing redundant work — should the two iterations conduct exactly the same computation?

```
//Apache37184 & Patch
//Project.java
public synchronized void
addBuildListener(BuildListener listener) {
+  if (!newListeners.contains(listener)) {
    newListeners.addElement(listener);
+  }
}

private void fireMessageLoggedEvent(...) {
  ...
  while (iter.hasNext()) {
    BuildListener listener = (BuildListener) iter.next();
    listener.messageLogged(event);
  }
}
```

Figure 6.5: A cross-iteration redundant bug in Apache

Second, is a (dynamic) loop problematic when it contains only few iterations that are redundant with each other?

Our answer to these two questions stick to one principle: there should be sufficient amount of redundant computation to make a loop likely root-cause for a user-perceived performance problem and to make itself worthwhile to get optimized by the developers. Consequently, for the first question, LDoctor takes a strict definition — only iterations that are doing exactly the same computation are considered redundant. Since one iterating may not contain too much computation, a weaker definition here may lead to many false positives. For the second question, we believe there should be a threshold. In our current prototype, when the number of distinct iterations is less than half of the total iterations, we consider the loop is cross-iteration redundant.

For example, Figure 6.5 shows a loop from Apache-Ant that contains cross-iteration redundancy. As we can see, under the problem triggering input, only several distinct listeners are contained inside the vector, and most of iterations of the loop inside function `fireMessageLoggedEvent` are doing exactly the same computation.

For cross-loop redundancy, we need to answer similar questions, especially how to judge whether two loop instances are doing redundant work — should

they contain exactly the same number of iterations and doing exactly the same computation in each iteration?

Our answers here are different from our answers above for cross-iteration redundancy analysis. We do not require two loop instances to conduct exactly the same computation to be considered redundant. The rationale is that a whole loop instance contains a lot of computation, much more than one iteration in general. Even if only part of its computation is redundant, it could still be the root-cause of a user-perceived performance problem and worth developers' attention.

In fact, in practice, we almost have never seen cases where different loop instances are doing exactly the same computation. For example, Figure 6.3 demonstrates a cross-loop redundancy problem in Mozilla. Here, the latter instances contain more iterations than previous instances. Figure 6.4 shows an example in Apache, The inner loop, which starts from line 8, searches from the beginning of a string `sb` for a target sub-string `s`. Since the outer loop, which starts on line 30, appends one character to `sb` in every iteration, every inner loop instance is doing computation that is similar, but not exactly the same, from its previous instance.

Detailed algorithm implementation The implementation of checking cross-iteration redundancy is straightforward. We will record a sequence of $< \mathtt{InstID}, \mathtt{Value} >$ pair for every monitored iteration, with each $\mathtt{InstID}$ representing a unique source instruction. We consider two iterations to be redundant, if their sequences are exactly the same. To make sure a loop contains sufficient redundant computation, we calculate a loop's *cross-iteration redundancy rate* — dividing the total number of iterations in the loop by the number of distinct iterations. The smaller the rate is, with 1 being the minimum possible value, the less cross-iteration redundancy the loop contains.

The implementation of checking cross-loop redundancy goes through several steps. First, for k dynamic instances of a static loop L that appear at run time, denoted as l_1, l_2, ..., l_k, we check whether redundancy exists between l_1 and l_2, l_2 and l_3, and so on. Second, we compute a *cross-loop redundancy rate* for L — dividing the number of redundant pairs by $k-1$. The smaller the rate

is, with 0 being the minimum possible value, the less cross-loop redundancy L contains. Here we only check redundancy between consecutive loop instances, because checking the redundancy between every pairs of loop instances would be very time consuming.

The key of this implementation is to judge whether two dynamic loop instances l_1 and l_2 are redundant or not. The challenge is that l_1 and l_2 may have executed different number of iterations; in different iteration, a different set of source instructions may have executed. Therefore, we cannot simply merge values from different source instructions and iterations together and compare two big data sequence. Instead, we decide to check the redundancy for each source instruction across l_1 and l_2 first, and then use the average *redundancy rate* of all source instructions as the *cross-loop redundancy rate* between l_1 and l_2.

We calculate the redundancy for one source instruction I by normalizing the edit-distance between the two sequences of values returned by I in the two loop instances. The exact formula is the following:

$$\mathtt{Redundancy}(\mathrm{I}) = \frac{\mathtt{dist}(\mathrm{SeqA}, \mathrm{SeqB}) - (\mathtt{len}(\mathrm{SeqA}) - \mathtt{len}(\mathrm{SeqB}))}{\mathtt{len}(\mathrm{SeqB})}$$

Here, SeqA and SeqB represent the two value sequences corresponding to I from two loop instances, with SeqA being the longer sequence. $\mathtt{dist}$ means edit distance, and $\mathtt{len}$ means the length of a value sequence. Since the edit distance is at least the length-difference between the two sequences and at most the length of the longer sequence, we use the subtraction and division shown in the formula above to normalize the redundancy value.

6.3.2.4 Dynamic performance optimization: sampling

Recording values returned by every source instructions would lead to huge runtime time. To lower the overhead, we use random sampling to reduce the number of instructions that we track at run time. Due to the repetitive nature of performance bugs, we will still be able to recognize redundant computation as long as the sampling rate is not too sparse (we will evaluate this in Section

6.4). Our sampling scheme requires almost no changes to our redundancy identification algorithm discussed in Section 6.3.2.3.

Cross-iteration redundancy analysis Our high-level sampling strategy is straightforward: randomly decide at the beginning of every iteration whether to track the values returned by source instructions in this iteration.

The implementation is similar with previous sampling work [56, 57]. Specifically, we create a clone of the original loop iteration code, including functions called by the loop directly or indirectly, and insert value-recording instructions along the cloned copy. We then insert a code snippet that conducts random decision to the beginning of a loop iteration. Two variables `CurrentID`, which is initialized as 0, and `NextSampleID`, which is initialized by a random integer, are maintained in this code snippet. `CurrentID` is increased by 1 for each iteration. When it matches `NextSampleID`, the control flow jumps to the value-recording clone of the loop iteration and the `NextSampleID` is increased by a random value. Different sampling sparsity setting will determine the range from which the random value is generated.

Cross-loop redundancy analysis At high level, we randomly decide at the beginning of every loop instance whether to track values for this instance. Since we will need to compare two consecutive loop instances for redundancy, once we decide to sample one loop instance, we will make sure to sample the immediately next loop instance too.

The implementation is similar with that for cross-iteration redundancy analysis. The only difference includes: (1) clone is made for the whole loop and the sampling control is done in the pre-header of the loops; (2) sampling is conducted when `CurrentID` equals either `NextSampleID` or `NextSampleID`+1, with `NextSampleID` increased by a random value in the latter case.

Handling recursive functions We also conduct sampling to our redundancy analysis for recursive functions. For a recursive function, we first create an instrumented clone copy of the whole function body. We then add a sampling-control code snippet at the entry point of the function, where `CurrentID` and `NextSampleID` are maintained to decide whether to execute the original function body or the instrumented value-recording clone copy. We also create

clones for all the callee functions of the recursive function under study, so that the sampling decision can be correctly conducted throughout the call chain.

6.3.2.5 Static analysis for performance optimization

We conduct a series of static analysis to reduce the number of instructions we need to monitor.

First, we identify and avoid monitoring memory reads whose return values can be statically proved to not change throughout one loop instance (i.e., for cross-iteration redundancy analysis) or multiple loop instances (i.e., for cross-loop redundancy analysis). Since we implement LDoctor at LLVM byte-code level, a major part of this analysis is already done by LLVM, which lifts loop-invariant memory accesses out of loops. The only extra analysis we did is to prune memory reads whose reading address is loop invariant, and there are no writes inside the loop which are possibly conducted on the same address. For example, the read address of `aNode.localName` and `aNode.namespaceURI` in Figure 6.1 is loop-invariant, and there are no write conducted on the same address. We do not need to record these two memory reads during cross-iteration redundancy analysis.

Second, we identify and avoid monitoring some memory reads whose return values can be statically proved to be different throughout one loop instance in cross-iteration redundancy analysis. Specifically, for read on loop induction variable, such as i for loop at line 11 of Figure 6.4, we also know for sure that their return values are different in different iterations. If source instructions include read on loop induction variables, we know that there could not be cross-iteration redundancy. We use scalar evolution analysis provided by LLVM to identify induction variables and avoid tracking their values.

Third, sometimes we only record the memory-address range of a sequence of memory read, instead of the value returned by every read, in cross-loop redundancy analysis. The loop at line 11 in Figure 6.4 shows an example. The content of array `source` is not heavily modified throughout the outer-loop, which starts at line 30 in the Figure 6.4. Therefore, to check whether different inner loop instances read similar sequence of array data, we only need to record

the starting and ending array index touched by each inner loop, significantly reduce the monitoring overhead. To accomplish this optimization, we again leverage the scalar evolution analysis provided by LLVM. The scalar evolution analysis tells us whether the address of a memory read instruction is a loop induction variable. For example, the address for memory read `source[i]` is added by one in each loop iteration, so it is a loop induction variable. From the scalar evolution analysis, we know that the starting address is `source` plus the initial value of `i`, and the ending address is source plus the ending value of `i`.

6.3.2.6 Fix strategy recommendation

As discussed in Section 6.2.2, extra analysis is needed to decide whether batching or memoization should be suggested to fix a loop that conducts redundant computation.

For cross-iteration redundancy, batching is often used towards batching I/O related operations, based on our empirical study. Therefore, we treat I/O related redundancy separately. Specifically, when the only side effect of a loop is I/O operations and the same statement(s) is executed in every loop iteration, we report this as I/O related redundancy problem and suggest batching as a potential fix strategy.

For cross-loop redundancy, whether to use memoization or batching often depends on which strategy is cheaper to use. LDoctor uses a simple heuristic. If the side effect of each loop instance is to update a constant number of memory locations, like the buggy loop in Figure 6.3 and Figure 6.4, we recommend memoization. Instead, if the side effect is updating an sequence of memory locations, with the number of locations increasing with the workload, memoization is unlikely to help save much computation.

6.4 Evaluation

6.4.1 Methodology

Implementation and Platform We implement LDoctor in LLVM-3.4.2 [52], and conduct our experiments on a i7-960 machine, with Linux 3.11 kernel.

BugID	KLOC	P. L.	Root Cause	Fix
Mozilla347306	88	C	0*1?	C
Mozilla416628	105	C	0*1?	C
Mozilla490742	N/A	JS	C-I	B
Mozilla35294	N/A	C++	C-L	B
Mozilla477564	N/A	JS	C-L	M
MySQL27287	995	C++	0*1?,C-L	C
MySQL15811	1127	C++	C-L	M
Apache32546	N/A	Java	C-I	B
Apache37184	N/A	Java	C-I	M
Apache29742	N/A	Java	C-L	B
Apache34464	N/A	Java	C-L	M
Apache47223	N/A	Java	C-L	B
GCC46401	5521	C	[0|1]*	S
GCC1687	2099	C	C-I	M
GCC27733	3217	C	C-I	M
GCC8805	2538	C	C-L	B
GCC21430	3844	C	C-L	M
GCC12322	2341	C	1*	S

Table 6.3: Benchmark information. N/A: we skip the size of benchmarks that are extracted from real-world applications. Root cause "C-I" is short for cross-iteration redundancy. Root cause "C-L" is short for cross-loop redundancy. C, B, M, and S represent different fix strategies, as discussed in Table 6.2.

Benchmarks We use 18 out of the 45 bugs listed in Table 6.2 as our evaluation benchmarks. Among these 18, seven are extracted from Java or JavaScript programs and re-implemented in C++, as LDoctor currently only handles C/C++ programs; one is extracted from a very old version of Mozilla. The remaining bugs listed in Table 6.2 are much more difficult to use as benchmarks, either because they depend on special hardware/software environment or because they involve too complex data structures to extract. Overall, these 18 bugs cover a wide variety of performance root causes, as shown in Table 6.3.

Metrics Our experiments are designed to evaluate LDoctor from three main aspects:

- Coverage. Given our benchmark suite that covers a wide variety of real-world root causes, can LDoctor identify all those root causes?

BugID	Reported Root Cause	Fix Suggestion
Mozilla347306	✓	✓
Mozilla416628	✓	✓
Mozilla490742	✓	✓
Mozilla35294	✓	✓
Mozilla477564	✓	✓
MySQL27287	✓	✗
MySQL15811	✓	✓
Apache32546	✓	✓
Apache37184	✓	✓
Apache29742	✓	✓
Apache34464	✓	✓
Apache47223	✓	✓
GCC46401	✓	✓
GCC1687	✓	✓
GCC27733	✓	✓
GCC8805	✓	✓
GCC21430	✓	✓
GCC12322	✓	✗

Table 6.4: Coverage Results.

- Accuracy. When analyzing non-buggy loops, will LDoctor generate any false positives?
- Performance. What is the runtime overhead of LDoctor?

Evaluation settings The imagined usage scenario of LDoctor is that one will apply LDoctor to identify detailed root causes and provide fix-strategy suggestion for a small number of suspicious loops that are most correlated with the specific performance problem.

Our evaluation uses the statistical performance diagnosis tool discussed in Chapter 5 to process a performance problem and identify one or a few suspicious loops for LDoctor to analyze. For 14 out of the 18 benchmarks, statistical performance debugging identifies the real root-cause loop as the top ranked suspicious loop. For the remaining benchmarks, the real root-cause loops are ranked number 2, 2, 4, and more than 5.

To evaluate the coverage, accuracy, and performance of LDoctor, we mainly conduct three sets of evaluation. First, we apply LDoctor to the real root-cause loop to see if LDoctor can correctly identify the root-cause category and pro-

vide correct fix-strategy suggestion. Second, we apply statistical performance debugging (Chapter 5) to all our benchmarks and apply LDoctor to the top 5 ranked loops[6] to see how accurate LDoctor is. Third, we evaluate the runtime performance of applying LDoctor to the real root-cause loop.

For all benchmarks we use, real-world users have provided at least one problem-triggering input in their on-line bug bugs. We use these inputs in our runtime analysis.

As discussed in Section 6.3, our analysis contains several configurable thresholds. In our evaluation, we use 0.001 as the *resultless rate* threshold for identifying 0*1? loops, 0.01 as the *resultless rate* threshold for identifying [0|1]* loops, 0.9 as the *cross-loop redundancy rate*, and 2 as the *cross-iteration redundancy rate* (i.e., the number of distinct iterations is less than half of the total iterations).

All the analysis and performance results presented below regarding cross-loop analysis is obtained using 1/100 sampling rate; all the results regarding cross-iteration analysis is obtained using 1/1000 sampling rate. We use sparser sampling rate in the latter case, because there tend to be more loop iterations than loop instances. All our diagnosis results require only **one** run under the problem-triggering input.

6.4.2 Coverage results

Overall, LDoctor provides good diagnosis coverage, as shown in Table 6.4. LDoctor identifies the correct root cause for **all** 18 benchmarks, and suggests fix strategies that exactly match what developers took in practice for 16 out of 18 cases. There are only two cases where LDoctor fails to suggest the fix strategy that developers used. For MySQL#27287, the root-cause loop is both cross-loop redundant and 0*1? inefficient. LDoctor suggests both changing data structures and memoization as fix strategies. In practice, the developers find a new data structure that can eliminate both root causes. For GCC#12322, LDoctor correctly tells that the loop under study does not contain any form of inefficiency and produce results in every iteration, and hence fails to suggest

[6]Some extracted benchmarks have fewer than 5 loops. We simply apply LDoctor to all loops in these cases.

BugID	0*1?	[0|1]*	C-I_b	C-I_m	C-L	Total
Mozilla347306	-	-	-	-	-	-
Mozilla416628	-	-	-	-	-	-
Mozilla490742	-	-	-	-	-	-
Mozilla35294	-	-	-	-	-	-
Mozilla477564	-	-	-	-	-	-
MySQL27287	-	0_1	-	-	-	0_1
MySQL15811	-	-	-	-	-	-
Apache32546	-	-	-	-	-	-
Apache37184	-	-	-	-	-	-
Apache29742	-	-	-	-	-	-
Apache34464	-	-	-	-	-	-
Apache47223	-	-	-	-	-	-
GCC46401	-	0_1	-	-	-	0_1
GCC1687	-	-	-	-	-	-
GCC27733	-	-	-	-	-	-
GCC8805	-	0_2	-	-	-	0_2
GCC21430	0_1	0_3	-	0_1	0_1	0_6
GCC12322	0_1	0_1	-	0_1	0_1	0_4

Table 6.5: False positives of LDoctor, when applying to top 5 loops reported by statistical performance diagnosis for each benchmark. '-' represents zero false positive. Other cells report real false positives and benign false positives, which is in the subscript.

any fix strategy. In practice, GCC developers decide to skip the loop, which will cause some programs compiled by GCC to be less performance-optimal than before. However, GCC developers feel that it is worthwhile considering the performance impact of the original loop to the GCC compilation process. Providing this type of fix strategy suggestion goes beyond the capability of LDoctor.

6.4.3 Accuracy results

As shown in Table 6.5, LDoctor is accurate, having 0 real false positive and 14 benign false positives for all the top 5 loops.

Here, benign false positives mean that the LDoctor analysis result is true — some loops are indeed cross-iteration/loop redundant or indeed producing results in only a small portion of all the iterations. However, those problems are *not* fixed by developers in their performance patches.

There are several reasons for these benign performance problems. The main reason is that they are not the main contributor to the performance problem perceived by the users. This happens to 12 out of the 14 benign cases. In fact, this is not really a problem for LDoctor in real usage scenarios, because statistical debugging can accurately tell that these loops are not top contributors to the performance problems. The remaining two cases happen when fixing the identified redundant/resultless problems are very difficult and hence developers decide not to fix them.

The accuracy of LDoctor benefits from its runtime analysis. For example, 4 benchmarks contain loops that only generate side-effect in the last iteration among their top 5 suspicious loops. However, these loops are actually not inefficient because the portion of resultless iterations is small.

The good accuracy of LDoctor can actually help improving the accuracy of identifying which loop is the root cause loop. For example, the real root-cause loop of Apache#34464 and GCC#46401 both rank number two by the statistical performance diagnosis tool. LDoctor can tell that the number one loops in both cases do not contain any form of inefficiency, resultless or redundancy. This result can potentially used to improve the accuracy of identifying root-cause loops.

6.4.4 Performance

As shown in Table 6.6, the performance of LDoctor is good. The overhead is consistently under or around 5% except for one benchmark, Mozilla#347306. We believe LDoctor is promising for potential production run usage. Of course, if we apply LDoctor to multiple loops simultaneously, the overhead will be higher. However, the current results are obtained by running the program only **once** under the problem-triggering workload. The sampling nature of LDoctor will allow us to keep the overhead low at the exchange of running the program for a couple of more times, if needed.

As we can also see from the table, our performance optimization discussed in Section 6.3.2.4 and 6.3.2.5 has contributed a lot to the good performance of LDoctor.

	LDoctor			w/o optimization	
BugID	**Resultless**	**C-L R.**	**C-I R.**	**C-L R.**	**C-I R.**
Mozilla347306	1.07%	22.40%	10.17%	304.37**X**	468.74**X**
Mozilla416628	0.80%	4.10%	2.99%	567.51**X**	85.6**X**
MySQL27287	~0	1.66%	-	109.55**X**	352.07**X**
MySQL15811	-	0.03%	-	227.04**X**	424.44**X**
GCC46401	3.12%	3.80%	5.95%	21.07**X**	38.44**X**
GCC1687	-	/	~0	/	142.29**X**
GCC27733	~0	/	4.73%	/	17.41**X**
GCC8805	-	~0	~0	2.22**X**	3.52**X**
GCC21430	-	5.46%	0.69%	107.20**X**	159.89**X**
GCC12322	-	1.75%	~0	21.07**X**	38.44**X**

Table 6.6: Runtime overhead of applying LDoctor to the buggy loop, with and without optimizations. Only results from non-extracted benchmarks are shown. -: static analysis can figure out the results and hence no dynamic analysis is conducted. /: not applicable.

Without sampling, while still applying our static optimization, our redundancy analysis would lead to over 100X slowdown for six benchmarks.

The buggy loops of MySQL#27287 and MySQL#15811 access arrays. After changing to tracking the initial and ending memory-access addresses of the array, instead of the content of the whole array accesses, the overhead is reduced from 11.77% to 1.66% for MySQL#27287, and from 20.46% to 0.03% for MySQL#15811 respectively (sampling is conducted consistently here).

The side-effect of the buggy loop for MySQL#15811 is to calculate the length of a string. The variable representing the length is an induction variable. The side-effect of the buggy loop for MySQL#27287 is to calculate the index of a searched target, and the variable representing the index is also an induction variable. We can reply on static analysis to figure out that these two loops are not cross-iteration redundant.

We also tried sampling with different sampling rates. For cross-loop redundancy, we also conduct experiments under sampling rate 1 out of 1000. Both runtime overhead and collected sample will be reduced. Mozilla#347306 is still the benchmark with largest overhead, but the overhead is reduced to 0.49%. For two benchmarks, GCC#8805 and GCC#12322, we cannot sample more than 10 loop instances and hence cannot draw strong conclusion about

their root-cause type. For all other benchmarks, we can still have more than 10 loop instances and get exactly the same root-cause analysis results presented above.

We also conduct cross-iteration redundant experiment under different sampling rates. When the sampling rate is 1 out of 100, there are two benchmarks, whose runtime overhead is larger than 30%. The overhead for Mozilla#347306 is 59.57%, and the overhead of GCC#21430 is 30.65%. When changing sampling rate to 1 out of 10000, Mozilla#347306 has the largest overhead, which is 4.47%. And Mozilla#347306 is the only one whose overhead is larger than 2%. Except for GCC#12322, all other benchmarks will have more than 100 iterations sampled, under the sample rate is 1 out of 10000. Consequently, the same root-cause analysis results will be reported for these benchmarks as the ones presented above.

6.5 Conclusion

Performance diagnosis is time consuming and also critical for complicated modern software. LDoctor tries to automatically pin-point the root cause of the most common type of real-world performance problems, inefficient loops, and provide fix-strategy suggestions to developers. It achieves the coverage, accuracy, and performance goal of performance diagnosis by leveraging (1) a comprehensive root-cause taxonomy; (2) a hybrid static-dynamic program analysis approach; and (3) customized random sampling that is a natural fit for performance diagnosis. Our evaluation shows that LDoctor can accurately identify detailed root causes of real-world inefficient loop problems and provide fix-strategy suggestions. Future work can further improve LDoctor by providing more detailed fix suggestions and providing more information to help diagnose and fix 1* loops.

Chapter 7

Conclusion

Performance bugs are software implementation mistakes that can cause inefficient execution. Performance bugs are common and severe, and they have already become one major source of software's performance problems. With the increasing complexity of modern software and workloads, the meager increases of single-core hardware performance, and pressing energy concerns, it is urgent to combat performance bugs.

This dissertation targets to improve the state-of-the-art performance bug fighting techniques. We start from an empirical study on real-world performance bugs in order to get a better understanding of performance bugs in Chapter 3. Inspired by our empirical study, we build a series of rule-based static checkers to identify previously unknown performance bugs in Chapter 4. We explore how to apply statistical debugging to diagnose user-perceived performance bugs in Chapter 5. Statistical debugging cannot answer all the questions, so we design a series of static-dynamic hybrid analysis for inefficient loops to provide fine-grained diagnosis information in Chapter 6.

In this chapter, we first summarize our study work and built techniques in Section 7.1. We then list a set of lessons learned over the course of our work in Section 7.2. Finally, we discuss directions for future research in Section 7.3.

7.1 Summary

This dissertation can be divided into three parts: performance bug understanding, performance bug detection, and performance failure diagnosis. We now summarize each part in turn.

7.1.1 Performance bug understanding

Like functional bugs, research on performance bugs should also be guided by empirical studies. Poor understanding of performance bugs is part of the causes of today's performance-bug problem. In order to improve the understanding of real-world performance bugs, we conduct the first empirical study on 110 real-world performance bugs randomly sampled from five open-source software suites (Apache, Chrome, GCC, Mozilla, and MySQL).

Following the lifetime of performance bugs, our study is mainly performed in four dimensions. We study the root causes of performance bugs, how they are introduced, how to expose them, and how to fix them. The main findings of our study include that (1) performance bugs have dominating root causes and fix strategies, which are highly correlated with each other; (2) workload mismatch and misunderstanding of APIs' performance features are two major reasons why performance bugs are introduced; and (3) around half of the studied performance bugs require inputs with both special features and large scales to manifest.

Our empirical study can guide future research on performance bugs, and it has already motivated our own bug detection and diagnosis projects.

7.1.2 Performance bug detection

Inspired by our empirical study, we hypothesize that efficiency rules widely exist in software, rule violations can be statically checked, and violations also widely exist. To test our hypothesis, we manually inspect final patches of fixed performance bugs from Apache, Mozilla, and MySQL in our studied performance-bug set. We extract efficiency rules from 25 bug patches and implement static checkers for these rules.

In total, our static checkers find 332 previously unknown Potential Performance Problems (PPPs) from the latest versions of Apache, Mozilla, and MySQL. Among them, 101 were inherited from the original buggy versions where final patches are applied. Tools are needed to help developers automatically and systematically find all similar bugs. 12 PPPs were introduced later. Tools are needed to help developers avoid making the same mistakes repeatedly. 219 PPPs are found by cross-application checking. There are generic rules

among different software. Our experimental results verify all our hypothesis. Rule-based performance-bug detection is a promising direction.

7.1.3 Performance failure diagnosis

Due to the preliminary tool support, many performance bugs escape from in-house performance testing and manifest in front of end users. After users report performance bugs, developers need to diagnose them and fix them. Diagnosing user-reported performance failure is another key aspect of fighting performance bugs.

We investigate the feasibility and design space to apply statistical debugging to performance failure diagnosis. After studying 65 user-reported performance bugs in our bug set, we find that the majority of performance bugs are observed through comparison, and that many user-file performance-bug reports contain not only bad inputs, but also similar and good inputs. Statistical debugging is a natural fit for user-reported performance bugs. We evaluate three types of widely used predicates and two representative statistical models. Our evaluation results show that branch predicates plus two statistical models can effectively diagnose user-reported performance failure. Basic model can help diagnose performance failure caused by incorrect branch decision, and ΔLDA model can identify inefficient loops. We apply sampling to performance failure diagnosis. Our experimental results show that a special nature of loop-related performance bugs allows sampling to lower runtime overhead without sacrificing diagnosis latency, which is very different from functional failure diagnosis.

We build LDoctor to further provide fine-grained diagnosis information for inefficient loops through two steps. We first figure out a root-cause taxonomy for common inefficient loops through a comprehensive study on 45 inefficient loops. Our taxonomy contains two major categories, resultless and redundancy, and several subcategories. Guided by our taxonomy, we then design a series of analysis for inefficient loops. Our analysis focuses its checking on suspicious loops pointed out by statistical debugging, hybridizes static and dynamic analysis to balance accuracy and performance, and relies on sampling and other designed optimization to further lower runtime overhead. We evaluate

LDoctor using 18 real-world inefficient loops. The evaluation results show that LDoctor can cover most root-cause subcategories, report few false positives, and bring a low runtime overhead.

7.2 Lessons Learned

In this section, we present a list of general lessons we learned while working on this dissertation.

Identifying performance failure is hard. Performance bugs do not have fail-stop symptoms. Both developers and end users face challenges in identifying performance failure runs. Performance bugs tend to hide longer in software than functional bugs (Section 3.8). Some performance bugs, which can be triggered by almost all inputs, can still escape from in-house performance testing (Section 3.6). Fearing that described symptoms are not convincing enough, end users sometimes use more than one comparison-based method when they file performance-bug reports (Section 5.2). Techniques targeting automatically identifying performance failure runs, such as automated oracles for performance bugs and performance assertion, are solely needed.

Similar mistakes can be made everywhere. Implementation mistakes are usually caused by developers' misunderstanding of programming languages, API, workload, documents, and so on. Before the misunderstanding is corrected, developers definitely would make similar mistakes in other places. Misunderstanding could also be shared among different developers, and the same mistakes could also be made by more than one developer. When fixing one bug, it is necessary to systematically check software to find similar bugs and fix them altogether.

Non-buggy codes can become buggy. Some performance bugs in our benchmark set are not born buggy. They become performance bugs, due to workload shift, code changes in other part of the software, or hardware changes. Periodic workload study, performance change impact analysis, and systematically profiling when porting software to new hardware are needed during software development and maintenance.

Static analysis is not accurate enough. Our experience shows that static analysis is not good enough when analyzing performance bugs, because some

codes are inefficient only under certain workloads. In Chapter 4, accurately checking some efficiency rules relies on runtime or workload information. Our detection techniques are only based on static checking, and this is one reason why we have false positives. In Chapter 6, static analysis can only identify static resultless types and conduct slicing. We need dynamic information about portions of resultless iterations and values of source instructions, so we build LDoctor based on static-dynamic hybrid approaches.

7.3 Future Work

Future work can explore how to combat performance bugs through different aspects from this thesis. The following are several potential research opportunities:

On-line workload monitoring. Our empirical study in Chapter 3 shows that many performance bugs are introduced due to developers' misunderstanding of workloads in reality. We think that monitoring workloads from production runs is a possible solution. Production-run techniques need to keep a low runtime overhead. How to collect accurate workload information from deployed software in a low overhead remains an open issue.

Improving the accuracy of static analysis by leveraging existing dynamic information. One lesson we learned is that static analysis alone is not accurate enough, and that dynamic information is needed (Section 7.2). Systems may have already recorded some dynamic information through logging or tracing. How to leverage existing dynamic information to improve the results of static analysis remains an open issue.

Applying sampling to existing in-house performance-bug techniques. Sampling can help lower runtime overhead, while recording dynamic information. Due to performance bugs' repetitive patterns, sampling does not hurt latency as much as functional bugs. In Chapter 5 and Chapter 6, we show how sampling helps statistical debugging and LDoctor. How to leverage sampling to make other existing in-house performance-bug detection and performance failure diagnosis techniques applicable in production runs remains an open issue.

Test input generation for performance bugs. In Chapter 3, we discuss that almost half of the studied performance bugs require inputs with both special features and large scales to manifest. Existing techniques are designed to generate inputs with good code coverage and focus only on special features. How to extend existing input-generation techniques with an emphasis on large scales remains an open issue. Another important problem during performance testing is to automatically judge whether a performance bug has occurred. How to leverage existing dynamic performance-bug detection techniques to build test oracles for performance bugs also remains an open issue.

Performance-aware annotation system. Our study in Chapter 3 shows that performance-aware annotations, which can help developers maintain and communicate APIs' performance features, can help avoid performance bugs. How to automatically identify performance features, such as the existence of lock or IO, through program analysis or document mining on existing software, remains an open issue.

7.4 Closing Words

Performance bugs are software implementation mistakes that can slow down the program. With new software and hardware trends and pressing energy concerns, it is very important to fight performance bugs. We hope that this dissertation can help researchers and developers by providing a better understanding of real-world performance bugs. We also hope that this dissertation can provide hints and implications for developers when they try to identify previously unknown performance bugs and diagnose user-perceived performance failure.

References

[1] http://sourceware.org/binutils/docs/gprof/.

[2] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. 2010. Performance analysis of idle programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 739–753. OOPSLA '10, New York, NY, USA: ACM.

[3] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. 2011. Fault-localization using dynamic slicing and change impact analysis. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 520–523. ASE '11, Washington, DC, USA: IEEE Computer Society.

[4] David Andrzejewski, Anne Mulhern, Ben Liblit, and Xiaojin Zhu. 2007. Statistical debugging using latent topic models. In *Proceedings of the 18th European Conference on Machine Learning*, 6–17. ECML '07, Berlin, Heidelberg: Springer-Verlag.

[5] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run software failure diagnosis via hardware performance counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 101–112. ASPLOS '13, New York, NY, USA: ACM.

[6] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems De-*

sign and Implementation, 307–320. OSDI'12, Berkeley, CA, USA: USENIX Association.

[7] Woongki Baek, and Trishul M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 198–209. PLDI '10, New York, NY, USA: ACM.

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

[9] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. 2013. A bloat-aware design for big data applications. In *Proceedings of the 2013 International Symposium on Memory Management*, 119–130. ISMM '13, New York, NY, USA: ACM.

[10] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, 463–473. ICSE '09, Washington, DC, USA: IEEE Computer Society.

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 209–224. OSDI'08, Berkeley, CA, USA: USENIX Association.

[12] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, 1001–1012. ICSE 2014, New York, NY, USA: ACM.

[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. In

Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, 265–278. ASPLOS XVI, New York, NY, USA: ACM.

[14] Andy Chou, Benjamin Chelf, Dawson Engler, and Mark Heinrich. 2000. Using meta-level compilation to check flash protocol code. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 59–70. ASPLOS IX, New York, NY, USA: ACM.

[15] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 73–88. SOSP '01, New York, NY, USA: ACM.

[16] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 89–98. PLDI '12, New York, NY, USA: ACM.

[17] Emilio Coppa, Camil Demetrescu, Irene Finocchi, and Romolo Marotta. 2014. Estimating the empirical cost function of routines with dynamic workloads. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 230:230–230:239. CGO '14, New York, NY, USA: ACM.

[18] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. 2013. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 329–342. ASPLOS '13, New York, NY, USA: ACM.

[19] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2011. Mining hot calling contexts in small space. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 516–527. PLDI '11, New York, NY, USA: ACM.

[20] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A cost-aware logging mechanism for performance diagnosis. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 139–150. Santa Clara, CA: USENIX Association.

[21] Amer Diwan, Matthias Hauswirth, Todd Mytkowicz, and Peter F. Sweeney. 2011. Traceanalyzer: a system for processing performance traces. *Softw., Pract. Exper.* 41(3):267–282.

[22] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2008. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 59–70. SIGSOFT '08/FSE-16, New York, NY, USA: ACM.

[23] Robert F. Dugan. 2004. In *Proceedings of the 4th International Workshop on Software and Performance*, 37–48. WOSP '04, New York, NY, USA: ACM.

[24] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 57–72. SOSP '01, New York, NY, USA: ACM.

[25] Lu Fang, Liang Dou, and Guoqing Xu. 2015. Perfblower: Quickly detecting memory-related performance problems via amplification. In *Proceedings of the 29th European Conference on Object-Oriented Programming*. ECOOP'15.

[26] Rodrigo Fonseca, Michael J. Freedman, and George Porter. 2010. Experiences with tracing causality in networked services. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, 10–10. INM/WREN'10, Berkeley, CA, USA: USENIX Association.

[27] Fortify. HP Fortify Static Code Analyzer (SCA). http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/.

[28] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. 2010. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 175–190. OOPSLA '10, New York, NY, USA: ACM.

[29] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering*. ICSE '15.

[30] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 103–116. SOSP '09, New York, NY, USA: ACM.

[31] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 213–223. PLDI '05, New York, NY, USA: ACM.

[32] Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering*, 156–166. ICSE '12, Piscataway, NJ, USA: IEEE Press.

[33] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. 2009. Speed: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, 127–139. POPL '09, New York, NY, USA: ACM.

[34] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 263–272. ASE '05, New York, NY, USA: ACM.

[35] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, 145–155. ICSE '12, Piscataway, NJ, USA: IEEE Press.

[36] Sudheendra Hangal, and Monica S. Lam. 2002. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, 291–301. ICSE '02, New York, NY, USA: ACM.

[37] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.* 12(1): 26–60.

[38] David Hovemeyer, and William Pugh. 2004. Finding bugs is easy. *SIGPLAN Not.* 39(12):92–106.

[39] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, 60–71. ICSE 2014, New York, NY, USA: ACM.

[40] InfoWorld. Top 10 open source hall of famers. http://www.infoworld.com/d/open-source/top-10-open-source-hall-famers-848.

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

Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, 77–88. PLDI '12, New York, NY, USA: ACM.

[42] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 389–400. PLDI '11, New York, NY, USA: ACM.

[43] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 241–255. OOPSLA '10, New York, NY, USA: ACM.

[44] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 221–236. OSDI'12, Berkeley, CA, USA: USENIX Association.

[45] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, 467–477. ICSE '02, New York, NY, USA: ACM.

[46] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch me if you can: Performance bug detection in the wild. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 155–170. OOPSLA '11, New York, NY, USA: ACM.

[47] Jyoti Bansal. Why is my state's aca healthcare exchange site slow? http://blog.appdynamics.com/news/why-is-my-states-aca-healthcare-exchange-site-slow/.

[48] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2010. Black-box problem diagnosis in parallel file systems. In *Proceedings of*

the 8th USENIX Conference on File and Storage Technologies, 4–4. FAST'10, Berkeley, CA, USA: USENIX Association.

[49] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. 2010. Finding latent performance bugs in systems implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 17–26. FSE '10, New York, NY, USA: ACM.

[50] Chung Hwan Kim, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Xiangyu Zhang, and Dongyan Xu. 2014. Introperf: Transparent context-sensitive multi-layer performance inference using system stack traces. *SIGMETRICS Perform. Eval. Rev.* 42(1):235–247.

[51] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 161–176. OSDI '06, Berkeley, CA, USA: USENIX Association.

[52] Chris Lattner, and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 75–. CGO '04, Washington, DC, USA: IEEE Computer Society.

[53] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, 20–20. OSDI'04, Berkeley, CA, USA: USENIX Association.

[54] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of*

the 1st Workshop on Architectural and System Support for Improving Software Dependability, 25–33. ASID '06, New York, NY, USA: ACM.

[55] Zhenmin Li, and Yuanyuan Zhou. 2005. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 306–315. ESEC/FSE-13, New York, NY, USA: ACM.

[56] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 141–154. PLDI '03, New York, NY, USA: ACM.

[57] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 15–26. PLDI '05, New York, NY, USA: ACM.

[58] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: saving DRAM refresh-power through critical data partitioning. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 213–224. ASPLOS XVI, New York, NY, USA: ACM.

[59] Tongping Liu, and Emery D. Berger. 2011. Sheriff: Precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 3–18. OOPSLA '11, New York, NY, USA: ACM.

[60] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, 1013–1024. ICSE 2014, New York, NY, USA: ACM.

[61] V. Benjamin Livshits, and Thomas Zimmermann. 2005. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 296–305. ESEC/FSE-13, New York, NY, USA: ACM.

[62] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2014. A study of linux file system evolution. *Trans. Storage* 10(1):3:1–3:32.

[63] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 329–339. ASPLOS XIII, New York, NY, USA: ACM.

[64] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 190–200. PLDI '05, New York, NY, USA: ACM.

[65] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic editing: Generating program transformations from an example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 329–342. PLDI '11, New York, NY, USA: ACM.

[66] Microsoft. MSDN SAL annotations. http://msdn2.microsoft.com/en-us/library/ms235402.aspx.

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

[68] Glen Emerson Morris. 2004. Lessons from the colorado benefits management system disaster. www.ad-mkt-review.com/publichtml/air/ai200411.html.

[69] Gilles Muller, Yoann Padioleau, Julia L. Lawall, and René Rydhof Hansen. 2006. Semantic patches considered helpful. *SIGOPS Oper. Syst. Rev.* 40(3): 90–92.

[70] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the accuracy of java profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 187–197. PLDI '10, New York, NY, USA: ACM.

[71] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 675–690. ASPLOS '15, New York, NY, USA: ACM.

[72] Khanh Nguyen, and Guoqing Xu. 2013. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 268–278. ESEC/FSE 2013, New York, NY, USA: ACM.

[73] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*.

[74] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, 237–246. MSR '13, Piscataway, NJ, USA: IEEE Press.

[75] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, 562–571. ICSE '13, Piscataway, NJ, USA: IEEE Press.

[76] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 369–378. PLDI 2015, New York, NY, USA: ACM.

[77] OProfile. OProfile – A System Profiler for Linux. http://oprofile.sourceforge.net.

[78] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 305–318. ASPLOS XVI, New York, NY, USA: ACM.

[79] Sharon E. Perl, and William E. Weihl. 1993. Performance assertion checking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, 134–145. SOSP '93, New York, NY, USA: ACM.

[80] Michael Pradel, Markus Huggler, and Thomas R. Gross. 2014. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 13–25. ISSTA 2014, New York, NY, USA: ACM.

[81] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. 2014. Eventbreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 33–47. OOPSLA '14, New York, NY, USA: ACM.

[82] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 107–120. OSDI'12, Berkeley, CA, USA: USENIX Association.

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

[84] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. 2010. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 47–56. PPoPP '10, New York, NY, USA: ACM.

[85] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 43–56. NSDI'11, Berkeley, CA, USA: USENIX Association.

[86] Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering*, 56–66. ICSE '09, Washington, DC, USA: IEEE Computer Society.

[87] Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. 2008. Jolt: Lightweight dynamic analysis and removal of object churn. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, 127–142. OOPSLA '08, New York, NY, USA: ACM.

[88] Richard L. Sites. Identifying dark latency.

[89] Connie U. Smith, and Lloyd G. Williams. 2000. Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance*.

[90] Linhai Song, and Shan Lu. 2014. Statistical debugging for real-world performance problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 561–578. OOPSLA '14, New York, NY, USA: ACM.

[91] Stefan Bodewig. Bug 45396: There is no hint in the javadocs. https://issues.apache.org/ bugzilla/show_bug.cgi?id=45396#c4.

[92] Mark Sullivan, and Ram Chillarege. 1992. A comparison of software defects in database management systems and operating systems. In *FTCS*.

[93] L. Torvalds. Sparse - a semantic parser for c. http://www.kernel.org/pub/software/devel/sparse/.

[94] Jeffrey S. Vetter, and Patrick H. Worley. 2002. Asserting performance expectations. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 1–13. SC '02, Los Alamitos, CA, USA: IEEE Computer Society Press.

[95] wikipedia. Chi-squared test. http://en.wikipedia.org/wiki/Chi-squared_test.

[96] Wikipedia. Z-test. http://en.wikipedia.org/wiki/Z-test.

[97] Xusheng Xiao, Shi Han, Tao Xie, and Dongmei Zhang. 2013. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 90–100. ISSTA 2013, New York, NY, USA: ACM.

[98] Guoqing Xu. 2012. Finding reusable data structures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 1017–1034. OOPSLA '12, New York, NY, USA: ACM.

[99] Guoqing Xu. 2013. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 111–130. OOPSLA '13, New York, NY, USA: ACM.

[100] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 419–430. PLDI '09, New York, NY, USA: ACM.

[101] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. Leakchaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 270–282. PLDI '11, New York, NY, USA: ACM.

[102] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 174–186. PLDI '10, New York, NY, USA: ACM.

[103] Guoqing Xu, and Atanas Rountev. 2010. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 160–173. PLDI '10, New York, NY, USA: ACM.

[104] Guoqing Xu, and Atanas Rountev. 2013. Precise memory leak detection for java software using container profiling. *ACM Trans. Softw. Eng. Methodol.* 22(3):17:1–17:28.

[105] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 117–132. SOSP '09, New York, NY, USA: ACM.

[106] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. 2014. Comprehending performance from real-world execution traces: A device-driver case. In *Proceedings of the 19th International Conference on Architectural Support*

for Programming Languages and Operating Systems, 193–206. ASPLOS '14, New York, NY, USA: ACM.

[107] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, 199–208. MSR '12, Piscataway, NJ, USA: IEEE Press.

[108] Dmitrijs Zaparanuks, and Matthias Hauswirth. 2012. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 67–76. PLDI '12, New York, NY, USA: ACM.

[109] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1–10. SIGSOFT '02/FSE-10, New York, NY, USA: ACM.

[110] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. 2006. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 263–271. PLDI '06, New York, NY, USA: ACM.