Statistical Debugging for Real-World Performance Problems

Linhai Song Shan Lu
University of Wisconsin–Madison
{songlh, shanlu}@cs.wisc.edu

Abstract

Design and implementation defects that lead to inefficient computation widely exist in software. These defects are difficult to avoid and discover. They lead to severe performance degradation and energy waste during production runs, and are becoming increasingly critical with the meager increase of single-core hardware performance and the increasing concerns about energy constraints. Effective tools that diagnose performance problems and point out the inefficiency root cause is sorely needed.

The state of the art of performance diagnosis is preliminary. Profiling can tell where computation resources are spent, but not where and why the resources are wasted. 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 paper, 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 run-time performance diagnosis without extending the diagnosis latency.

1. Introduction

1.1 Motivation

Implementation or design defects in software can lead to inefficient computation, causing unnecessary performance losses at run time. Previous study has shown that this type of performance-related software defects[1] widely exist in real-world [9, 18, 21, 28, 34]. They are difficult for developers to avoid due to the lack of performance documentation of APIs and the quickly changing workload of modern software [18]. A lot of performance bugs escape the in-house testing and manifest during production runs, causing severe performance degradation and significant energy waste in the field [18]. Making things worse, the negative impact of these performance problems is getting increasingly significant, with the increasing complexity of modern software and workload, the meager increases of single-core hardware performance, and the pressing energy concerns. Effective techniques to diagnose real-world performance problems are sorely needed.

```
void start_bulk_insert(ha_rows rows)
{
  ...
-  if (!rows)
-    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 1: A real-world performance bug in MySQL (the '-' and '+' demonstrate the patch)

The state of practice of performance diagnosis is preliminary. The most commonly used and often the only available tool during diagnosis is profiler [1, 31]. 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 of performance problems.

Figure 1 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

[1] We will refer to these defects as performance bugs or performance problems interchangeably following previous work in this area [18, 21, 30].

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 is extremely slow. The final patch simply removes the unnecessary branch in Figure 1, 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 [7]; trace analysis techniques have been proposed to figure out the performance-causality relationship among system events and components [10, 40]. 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 [30], under-utilized data structures [38], and temporary object bloat [11, 36, 37], 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.

1.2 Can we learn from functional failure diagnosis?

Automated failure diagnosis has been studied for decades for functional bugs[2]. Many useful and generic techniques [14, 16, 17, 19, 22, 42] have been proposed. It would be nice if some of these techniques can work for diagnosing performance problems.

Statistical debugging is a well studied and effective failure-diagnosis technique for functional bugs [17, 19, 22]. It 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 the predicates that are most correlated with the failure, referred to as failure predictors. Whether it is useful for performance bugs is still an open question due to several challenges (or mysteries).

First, how to tell success runs from failure runs in the context of performance bugs. Clearly, the answer to this question is the foundation for statistical debugging. Intuitively, it is much easier to answer this question for functional bugs than that for performance bugs. Crashes, incorrect outputs, and hangs are all clear symptoms of functional bug failures. Instead, given a slow execution, it is hard to tell whether the slowness is caused by large workload or the manifestation of a performance bug.

Second, how to obtain a set of inputs that drive success runs and failure runs for failure diagnosis. If the success runs and failure runs are completely different from each other, it is difficult to get good failure predictors; if there is only one failure run, it is also difficult to get good diagnosis results. Previous work on functional-bug failure diagnosis has studied how to generate good inputs and bad inputs that are similar with each other [42]. It is unclear how to do this for performance bugs, without answering the first question.

Third, what program properties (i.e., predicates) can reflect common root causes of performance problems. Different predicates are designed for statistical debugging that targets different types of functional bugs. For example, branch predicates, function-return predicates, and others are used for diagnosing sequential bugs [22, 23]; interleaving-related predicates are used for diagnosing concurrency bugs [6, 17]. It is unclear what are the common root causes of real-world performance bugs, not to mention what are the most useful predicates for performance diagnosis.

1.3 Contributions

This paper 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 answer the first two questions discussed in Section 1.2, we study how users notice and report performance problems, and how developers diagnose these problems, based on 65 real-world user-reported performance problems in five representative open-source applications (Apache, Chrome, GCC, Mozilla, and MySQL). Our study finds that (1) users notice the symptoms of most performance problems through a comparison-based approach (more than 80% of the cases); (2) many users' performance bug reports provide two sets of inputs that look similar with each other but have significantly different performance (about 60% of the cases); (3) performance diagnosis is time consuming, taking more than 100 days on average, and lacking tool support, taking more than 100 days on average even after getting the profiling results. These findings provide guidance and motivation for statistical debugging for performance problems, and the details are presented in Section 2.

A thorough study of statistical in-house performance diagnosis Motivated by the empirical study, we set up a statistical debugging framework for diagnosing performance problems. It includes a comprehensive set of design points — three representative predicates (branches, function returns, and scalar-pairs) and two different types of statistical models. We evaluate these different design points by experiments on 20 user-reported performance problems and man-

[2] Any software defects that lead to functional misbehavior, such as incorrect outputs, crashes, and hangs. They include semantic bugs, memory bugs, concurrency bugs, and so on.

ual 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, significantly improving the state of the art of performance diagnosis. More details are presented in Section 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 run-time 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 4.

2. Understanding Real-World Performance Problem Reporting and Diagnosis

This section aims to understand the performance diagnosis process in real world. Specifically, we will study the following aspects of performance diagnosis.

1. How users notice and report performance problems. This will help us understand the two key questions discussed in Section 1.2: (1) how to tell success runs from failure runs in the context of performance bugs; and (2) 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.

2.1 Methodology

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 1: Applications and bugs used in the study

The performance problems under this study include all user-reported performance problems from a real-world performance-bug benchmark suite collected by previous work [18]. We briefly discuss this baseline benchmark suite and our refinement below.

The baseline benchmarks [18] contain 110 fixed real-world performance bugs randomly sampled from five representative open-source software suites. These five software suites are all large and mature, with millions lines of codes and well maintained bug databases. They also provide a good coverage of different types of software projects, as shown in Table 1. The 110 bugs contained in this baseline suite are from on-line bug databases and are tagged by developers as performance bugs.

We cannot directly use this 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 remainder of this paper. The detailed distribution of these 65 bugs is shown in Table 1.

Caveats Similar with all previous characteristics studies, our findings and conclusions need to be considered with our methodology in mind. The applications in our study cover a variety of important software categories, workload, development background, and programming languages. However, there are still uncovered categories, such as scientific computing software and distributed systems.

The bugs in our study are collected from an earlier benchmark suite [18] 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.

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 diagnosis. Specifically, as discussed in Section 1.2, we have to solve two mysteries in or-

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	5	21
Comparing inputs with slightly different functionality	2	0	3	2	5	12
Comparison cross multiple code bases	8	3	8	5	4	28
Comparing the same input under same application's different versions	4	2	8	3	3	20
Comparing the same input under different applications	5	1	1	2	1	10
Not using comparison-based methods	3	1	0	9	1	14

Table 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)

	Apache	Chrome	GCC	Mozilla	MySQL	Total
Total # of bug reports	16	5	9	19	16	65
# of bad inputs provided						
1/?: One bad input provided	1	1	5	6	7	20
n/?: A set of bad inputs provided	15	4	4	13	9	45
# of good inputs provided						
?/0: No good input provided	7	2	2	12	4	27
?/1: One good input provided	1	0	3	0	3	7
?/n: A set of good inputs provided	8	3	4	7	9	31

Table 3: Inputs provided in users' bug reports (n: developers provide a way to generate a large number of inputs)

der to apply statistical debugging for performance diagnosis. How do users judge whether a slow execution is caused by large workload or inefficient implementation, telling success runs from failure runs? What information do users provide to convince developers that inefficient implementation exists? Answers to these questions will guide the design of suitable diagnosis approaches for real-world performance problems.

How are performance problems observed? As shown in Table 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 significantly different performance under 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, significantly different performances 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 suspicion about the performance problem 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 15 out of 28 cases, comparison results cross 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 "froze to crawl".

What information is available 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 bad performance as *bad inputs*; we refer to the inputs that are similar with some bad inputs but lead to good performance as *good 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 (45 out of 65), users describe a category of inputs, instead of just one input, that can trigger the performance problem, as shown in Table 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 3. That is, users describe inputs that are similar with the bad inputs but have significantly better performance in all the 38 bug reports where "comparison within one code base" is used to observe the performance problem. Furthermore, in 31 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) is always much faster than that on tables using other types of charset settings (bad inputs). Note that, this is much rarer in functional bug reports, which is why tools are designed to automatically generate inputs that execute correctly and are similar with bad inputs, when diagnosing functional bug failures [42].

2.3 How developers diagnose performance problems

Our study shows that it takes 130 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 by previous empirical studies, the time to diagnose user-reported performance problems is slightly shorter than that for non-user-reported performance problems [18], and similar or longer than that of functional bugs [18, 26].

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.

2.4 Implications of the study

Implication 1 Performance bugs and functional bugs are observed in different ways. The symptom of a functional bug can be easily identified by looking at the failure run alone (e.g., crashes). In contrast, the manifestation of performance bugs often gets noticed through comparison.

Implication 2 It is fairly easy to tell success runs from failure runs following a performance-bug report filed by users. For all the bugs that we have studied, there are only two scenarios. In one scenario, which is more often, users provide sufficient information for developers to set up a comparison environment. The performance difference under comparison is huge. In the other scenario, which is less often, the symptom of the problematic run itself is dramatic (e.g., application completely freezes).

Implication 3 Statistical debugging is actually naturally suitable for diagnosing performance problems, because most performance bugs are observed through comparison and many performance bug reports already contain information about bad and good inputs that are similar with each other. Since most functional bugs are **not** observed through comparisons, previous statistical debugging for functional bugs often need to look for inputs that are similar with the bug triggering inputs yet produce correct execution results [42]. The quality of good inputs often significantly affects the accuracy of statistical debugging. Fortunately, this is much less an issue for diagnosing performance problems.

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

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 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.

3.1 Design

In general, statistical debugging [4, 6, 17, 19, 22, 23, 33] is an approach that uses statistical machine learning techniques to help failure diagnosis. It usually works in two steps. First, a set of run-time 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.

BugID	KLOC	Language	Static # of predicates			Static # of Loops	Reported Inputs (bad/good)
			Branch	Return	Scalar-pair		
Mozilla258793	3999	C++	385722	1126770	—	10016	n/0
Mozilla299742	3954	C++	385720	1136698	—	10016	1/0
Mozilla347306	88	C	31182	41592	247458	951	n/n
Mozilla411722	101	C	35994	45354	311874	1457	1/0
MySQL15811	1149	C++	13508	15678	—	760	n/n
MySQL26527	986	C++	90128	129402	—	4222	n/n
MySQL27287	995	C++	92316	119946	—	4683	n/n
MySQL40337	1191	C++	103686	139422	—	4510	n/n
MySQL42649	1164	C++	126822	156720	—	5688	n/n
MySQL44723	1164	C++	126822	156720	—	5688	1/1
Apache3278	N/A	Java	10	114	204	6	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	1798	C	183496	296058	4187586	6476	n/n
GCC8805	2174	C	207188	327804	4161012	7309	n/n
GCC15209	2177	C	192108	304800	3705558	7310	1/1
GCC21430	3002	C	238514	447510	3768078	8237	n/n
GCC46401	5233	C	337810	713532	5625606	15156	1/1
GCC12322	2174	C	177098	284484	3750912	6563	1/0

Table 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. The 1s and *n*s in the "Reported Inputs" column indicate how many bad/good inputs are reported by users.)

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 run-time 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 2. We discuss different predicate designs and statistical model designs below.

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 [22, 23].

Returns. There are a set of three return predicates for each function return point, tracking whether the return value is negative, zero, or positive [22, 23].

Scalar-pairs. There are three scalar-pair predicates for each pair of variables x and y, tracking whether x is smaller than, larger than, or equal to y [22, 23]. 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 [4, 19, 33].

Interleaving-related ones. Previous work on diagnosing concurrency bugs [17] 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. 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.

3.1.2 Statistical model designs

Most statistical debugging work [4, 17, 19, 22, 23, 33] uses statistical models that only care about whether a predicate is true for at least once during a run. We consider these as basic models. Relatively few previous work [5] uses much more sophisticated statistical models that consider how many times a predicate has been true in each run. We will discuss one such model, ΔLDA [5], below.

Basic model Although the exact models used by previous work [4, 19, 22, 23, 33] differ from each other, they mostly follow the same principle. That is, they think a good failure predictor should be true for many failure runs and should be false or not-observed in many success runs. They only consider 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 does not matter.

In our experimental evaluations, we use the predicate ranking formula proposed by CBI work [22, 23]. CBI statistical model works in two steps. First, it uses the *Increase* metric to filter out predicates that are clearly not failure predictors. This metric measures how a predicate being true increases the chance of execution failure comparing with this predicate merely being observed. Only predicates, whose *increase* values are strictly higher than 0 with certain confidence, will appear in the final ranking list. For each predicate P, $Increase(P)$ is calculated as follows:

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

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

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

$F(P)$ 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. $S(P)$ 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.

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

The final ranking is based on an *Importance* metric, which considers both the *Increase* metric and how often a predicate is observed true for at least once during a failure run. Roughly speaking, to get a high ranking, a predicate needs to be true for at least once in many failure runs and few success runs, and be false for at least once in few failure runs and many success runs.

ΔLDA model ΔLDA [5] model is derived from a famous machine learning model, called Latent Dirichlet Allocation (LDA) [8]. It has already been applied to model natural language documents, images, and others. Past work [5] has found that it is good at picking up weaker bug signals that cannot be picked up by the basic model. A significant difference between it and the basic model is that it considers how many times a predicate is true in each success and failure run. Imagine the following scenario — during a success run, predicate *P* is true for 10 times and false for 20 times; during a failure run, *P* is true for 20 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 predicting. The exact ranking formula of ΔLDA model is very complicated, and is skipped here.

Note that, it takes much more time to process the predicate statistics using the ΔLDA model than using the basic model. Most previous work on functional bug diagnosis has found the basic model to be sufficient [6, 17, 22, 23] and did not try ΔLDA style models.

3.2 Experimental evaluation

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 2, we have tried our best effort and successfully repeated 20 of them from four different C/C++/Java applications. We have failed to repeat the remaining 45 performance problems for this study mainly because they depend on special hardware platforms or very old libraries that are not available to us at the moment of the submission. The detailed information for the 20 performance problems used in our experiments is shown in Table 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 and other settings 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. As shown in Table 4, in many cases, users provide a methodology to generate a large number of good and bad inputs. We follow that methodology in those cases. In the remaining cases when users only provide one good/bad input, we randomly change the provided input and use the user-provided failure-symptom information to decide which inputs are good or bad. As discussed in Section 2, the process of judging which inputs are good or bad is straightforward.

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 [22, 23] to collect all these three types of predicates[3]. For C++ programs, we implement our own branch-predicate and return-predicate collection tools using PIN binary-instrumentation framework [27]. 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 3.3).

We also use *OProfile* [31] to get profiling results in our experiments. The "Profiler" column in Table 5 will show 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.

3.2.2 Results for basic model

Overall, a significant number (9 out of 20) of performance problems can be successfully diagnosed using the basic statistical model. Furthermore, in all these 9 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, and the scalar-pair predicate is the least useful. Function-return predicate is useful for diagnosing one performance problem from Apache, as shown in Figure 2. 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 the best failure predictor is the fact that `pthread_cond_timedwait` returns a positive value, which indicates 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`

[3] CBI [22, 23] 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.

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

Figure 2: An Apache bug diagnosed by Return

```
1  //ha_myisam.cc
2  /* don't enable row cache if too few rows */
3  if (! rows || (rows > MI_MIN_ROWS_TO_USE_WRITE_CACHE) )
4      mi_extra(...);
5  //mi_extra() will allocate write cache
6  //and zero-fill write cache
7  // fix is to remove zero-fill operation
8  ....
9  // in myisamdef.h:
10 // #define MI_MIN_ROWS_TO_USE_WRITE_CACHE 10
```

Figure 3: A MySQL bug diagnosed by Branch

and issue a signal even before the `notified` is initialized to be `false` on line 1 of Figure 2, 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 2 is only two lines away from the failure predictor. As another example, the top-ranked failure predictor for the MySQL bug shown in Figure 1 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 3 illustrates the performance problem reported in MySQL#44723. MySQL44723 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, 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

	# of candidate predicates			Basic model			ΔLDA model	Profiler	Developers' fix strategy
BugID	Branch	Return	S-pair	Branch	Return	S-pair	Branch		
Mozilla258793	64024	152724	/	✓$_1$(0)	-	/	-	-	Change branch condition
Mozilla299742	64089	150973	/	✓$_1$(0)	-	/	-	-	Change branch condition
Mozilla347306	6901	6729	30953	-	-	-	✓$_1$(1)	✓$_1$(7)	Remove the loop
Mozilla411722	8780	6889	34378	✓$_1$(1)	-	-	-	-	Optimize branch body
MySQL15811	1198	886	/	-	-	/	✓$_1$(.)	✓$_1$(.)	Remove the loop
MySQL26527	7443	7631	/	✓$_1$(0)	-	/	-	-	Change branch condition
MySQL27287	5377	5762	/	-	-	/	✓$_1$(0)	✓$_1$(0)	Remove the loop
MySQL40337	7547	8161	/	✓$_1$(1)	-	/	-	-	Change branch condition
MySQL42649	15920	11800	/	✓$_1$(.)	-	/	-	-	Optimize branch body
MySQL44723	10649	9130	/	✓$_1$(.)	-	/	-	-	Optimize branch body
Apache3278	7	57	102	-	✓$_1$(2)	-	-	-	Synchronization adjustment
Apache34464	17	23	203	-	-	-	✓$_3$(0)	✓$_5$(2)	Combine loop instances
Apache47223	17	15	235	-	-	-	✓$_1$(.)	✓$_1$(.)	Combine loop iterations
Apache32546	5	34	69	-	-	-	✓$_1$(11)	-	Combine loop iterations
GCC1687	22656	17829	427649	-	-	-	✓$_1$(.)	✓$_1$(.)	Combine loop instances
GCC8805	23991	20530	406567	-	-	-	✓$_4$(0)	-	Reduce # loop iterations
GCC15209	10324	10721	171789	✓$_1$(8)	-	-	-	-	Change branch condition
GCC21430	45489	51266	646962	-	-	-	✓$_1$(0)	✓$_1$(2)	Remove the loop
GCC46401	34378	38275	480594	-	-	-	✓$_2$(.)	✓$_5$(.)	Reduce # loop iterations
GCC12322	46715	38259	877628	-	-	-	-	-	Reduce # loop iterations

Table 5: Experimental results for in-house diagnosis (✓$_x$(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. -: none of the top five predictors are related to the root cause or no predicates reach the threshold of the statistical model.).

out of thousands or even hundreds of thousands of candidate predicates.

Comparing with the profiler For nine 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 9 cases, the root-cause functions are ranked from number 18 to number 2673 for 5 cases. In the other 4 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).

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 8 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 (3 cases). For the one case where a return predicate is the best failure predictor, the fix affects the return value of the corresponding function.

3.2.3 Results for ΔLDA model

Following the results from the basic statistical model, we tried statistical debugging using the ΔLDA model together with branch predicates. Specifically, we focus on predicates collected at loop-condition branches here.

As shown in Table 5, this new design point (ΔLDA plus loop-branch) well complements the statistical debugging designs discussed earlier (i.e., basic statistical model). In 10 out of 11 cases where the basic statistical model fails to identify good failure predictors, useful failure predictors are identified by the ΔLDA model.

In general, when a loop-branch predicate b is considered 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 seven cases, the loop ranked number one is exactly the root cause of computation inefficiency. Developers fix this problem by either (1) completely removing the inefficient loop from the program (indicated by "Remove the loop" in Table 5); or (2) batch-processing multiple iterations together to remove redundancy across loop iterations (indicated by "Combine # loop iterations" in Table 5).

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

Fix Categories	Apache	Chrome	GCC	Mozilla	MySQL	Total
Total # of loop-related bugs	10	4	7	12	9	42
Remove the loop	0	1	2	4	2	9
Reduce # loop instances (memoization across loop instances)	5	0	1	1	2	9
Reduce # loop iterations (reduce the workload of the loop)	0	0	4	2	2	8
Combine loop iterations (removing cross-iteration redundancies)	4	3	0	4	0	11
Others	1	0	0	1	3	5

Table 6: Fix strategies for loop-related bugs

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 more significant iteration-number increases from success runs to failure runs.

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 the ones discussed in Section 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 6 cases) or better (in 4 cases) 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 significant 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.

3.3 Manual inspection

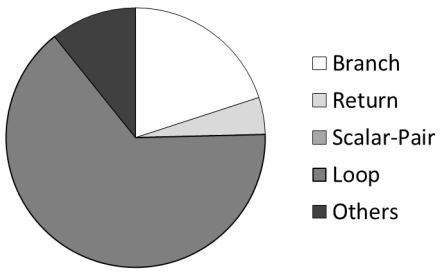

Figure 4: How different predicates work for diagnosing user-perceived performance bugs (manual inspection)

To achieve a more comprehensive understanding, we manually studied which predicate, if any, would help diagnose each of the 65 user-reported performance bugs in our benchmark set. The result is shown in Figure 4.

Assuming the basic statistical model, traditional predicates (i.e., branches, returns, and scalar-pairs) can diagnose 16 out of 65 performance problems. Among them, branch predicates are the most helpful, able to diagnose 13 performance problems; return predicates can diagnose 3 performance problems; scalar-pair predicates cannot diagnose any benchmark in our set.

Among the ones that cannot be diagnosed by the basic statistical model, 42 of them are caused by inefficient loops. We expect that the ΔLDA statistical model can identify root-cause related branch predicates (denoted as "loop" in Figure 4), as demonstrated by our experimental results in Section 3.2.3.

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

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 [4, 6, 17, 19, 22, 23, 33] 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 informa-

tion 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.

To guide future research on performance diagnosis, we further studied those 42 loop-related performance problems, and manually categorized their fix strategies, as shown in Table 6. We expect future performance diagnosis systems to use static or dynamic analysis to automatically figure out the detailed root causes, differentiate effects from causes, and suggest detailed fix strategies, after statistical debugging identifies root-cause loop candidates.

4. Production-run statistical debugging

In-house performance diagnosis discussed in Section 3 assumes that users file a comprehensive 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 run-time information. The key challenge of diagnosing these production-run failures includes how to collect production-run information with small overhead and how to conduct failure diagnosis with the small amount of automatically collected information. This section discusses this issue in the context of performance bugs.

4.1 Design

There are two key design goals for production-run performance diagnosis. First, the run-time overhead has to be low, not disturbing end users' normal operations. Second, the diagnosis capability has to remain high, accurately diagnosing a wide variety of problems.

Previous work on production-run functional bug diagnosis [6, 17, 22, 23] proposes to achieve these two goals by combining *sampling* techniques with statistical debugging. By randomly sampling predicates at run time, the overhead can be significantly decreased; by processing predicates collected from many failure and success runs together, the diagnosis capability can be maintained for the diagnosis of most functional bugs [6, 17, 22, 23]. We plan to follow this approach and apply it for production-run performance diagnosis.

Different from production-run functional failure diagnosis [6, 17, 22, 23], 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 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 paper 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 [22, 23] that can discover multiple failure predictors corresponding to different root causes mixed in one profile pool, as well as some failure bucketing techniques [13] 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.

4.2 Experimental evaluation

Our experimental evaluation aims to answer two 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. How many failure runs are needed to complete the failure diagnosis? 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.

4.2.1 Methodology

Benchmarks and inputs We reuse the same set of benchmarks shown in Table 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 and setting To sample return predicates, we directly use CBI [22, 23]. 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 leverage hardware performance counters following the methodology described in previous work [6]. Specifically, no program modification is made. Instead, 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).

In our experiments, we keep the sampling rate at roughly 1 out of 100. We first collect 1000 success-run profiles and 1000 failure-run profiles for diagnosis. We then try using only 10 success runs and 10 failure runs to see if we can achieve good diagnosis capability, low diagnosis latency, and low run-time overhead simultaneously when ΔLDA model is used. Since sampling is random, we have repeated our evaluation for several rounds to confirm that all the presented results are stable.

4.2.2 Results

BugID	Diagnosis capability	Run-time overhead
Mozilla258793	✓$_1$	1.81%
Mozilla299742	✓$_1$	7.52%
Mozilla347306	✓$_1$	3.01%
Mozilla411722	✓$_1$	3.35%
MySQL15811	✓$_1$	8.58 %
MySQL26527	✓$_1$	7.06 %
MySQL27287	✓$_1$	2.62 %
MySQL40337	✓$_1$	3.31 %
MySQL42649	✓$_1$	4.67 %
MySQL44723	✓$_1$	0.40 %
Apache3278	✓$_1$	3.22 %
Apache34464	✓$_3$	2.13 %
Apache47223	✓$_1$	2.77 %
Apache32546	✓$_1$	0.45 %
GCC1687	✓$_1$	1.01%
GCC8805	✓$_9$	2.66%
GCC15209	✓$_1$	3.41%
GCC21430	✓$_1$	1.73%
GCC46401	✓$_2$	2.51%
GCC12322	-	2.29%

Table 7: Sampling-based production-run performance diagnosis (we study whether the predicate and statistical model combination that worked for diagnosing a benchmark without sampling in Section 3 can still work with sampling)

Diagnosis capability As shown by Table 7, with sufficient runs (1000 success runs and 1000 failure runs), sampling did not degrade the diagnosis capability of statistical debugging. In all 20 cases, the root-cause related failure predictors discussed in Section 3.2 still rank high with sampling. In fact, in only one case (GCC8805), the ranking of the ideal failure predictor drops from number four to number nine due to the sampling effects.

Run-time overhead As also shown in Table 7, sampling significantly lowers the run-time overhead to always below 10%. In fact, the overhead is below 5% in all but three cases.

Diagnosis latency Diagnosis latency versus run-time overhead is a fundamental challenge facing sampling-based statistical debugging for functional bugs [17, 22, 23]. 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 [13]. However, this is a problem for software that is not widely deployed.

One unique and nice feature about performance bugs, especially loop-related performance bugs, is that their diagnosis latency is often much less sensitive to sampling rate than functional bugs, because the root-cause related predicates are often evaluated to be true for many times in one run, which is why the performance is poor.

To validate this hypothesis, we tried diagnosis using only 10 failure runs and 10 success runs, under the sampling rate of 1 out of 100. Our experiments show that the ten performance problems that can be successfully diagnosed by the ΔLDA model in Section 3 can still be diagnosed in this sampling-based setting. The only benchmark whose ideal predictor' ranking drops is GCC#8805. For all other benchmarks, the rankings are exactly the same with or without sampling, with just 10 failure runs. Consequently, sampling allows us to achieve both low run-time overhead ($<$10%) and low diagnosis latency, a feat that is almost impossible for sampling based functional bug diagnosis.

5. Related Works

5.1 Empirical study of performance bugs

Recently, several empirical studies have been conducted for real-world performance bugs. They all have different focuses. Some of them [41] compare the qualitative difference between performance bugs and non-performance bugs across impact, context, fix and fix validation; some of them [18] look at how performance bugs are introduced, how performance bugs manifest, and how performance bugs are fixed; some of them [25] focuses on performance bugs in smart-phone applications. Different from all previous studies, our study aims to provide guidance to performance problem diagnosis, and hence focuses on how performance problems are noticed and reported by end users.

A most recent study conducted by Nistor et. al. [29] is similar with our bug characteristics study (Section 2) in that it also finds that performance problems take long time to get diagnosed and the help from profilers is very limited. However, the similarity ends here. Different from our study, this recent work did 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,

which is not the focus of our study. In short, it does not aim to guide automated diagnosis of performance problems, and is hence different from our work.

5.2 Performance problem diagnosis

Diagnosis tools aim to identify root causes and suggest fix strategies when software failures happen. Tools have been proposed to diagnose certain type of performance problems.

X-ray [7] 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 pin-points the inputs or configuration entries that are most responsible for a performance problem, and help users to solve the performance issues by 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. Techniques discussed in our paper aim to help developers. We want to provide information to help developers change inefficient code and fix performance bugs. StackMine [15] automatically identifies certain call-stack patterns that are correlated with performance problems of event handlers. Yu et al. [40] automatically processes detailed system traces to help developers understand how performance impact propagates across system components, and what are the performance causality relationships among components and functions. All these diagnosis tools are very useful in practice, but have different focus from our work. They do not aim to identify source-code fine-granularity root causes of performance problems reported by end users.

Many techniques been proposed to diagnose performance problems in distributed systems [2, 12, 20, 32, 39]. These techniques often focus on identifying the faulty components/nodes or faulty interactions that lead to performance problems, which are different from our work.

5.3 Performance bug detection

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

Some tools [11, 36, 37] detect runtime bloat, a common performance problem in object-oriented applications. Xu et al. [38] targets low-utility data structures with unbalanced costs and benefits. Jin et al. [18] employ rule-based methods to detect performance bugs that violate efficiency rules that have been violated before. WAIT [3] focuses on bugs that block the application from making progress. Liu and Berger [24] build two tools to attack the false sharing problem in multi-threaded software. There are also tools that detect inefficient nested loops [30] and workload-dependent loops[35].

These bug-detection tools have different focus from our work. They do not focus on handling generic performance problems. The bug detection is also not guided by specific performance symptoms reported by end users.

6. 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 paper 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 method 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 future directions for future work on fine-granularity performance diagnosis.

Acknowledgments

We thank Professor Darko Marinov and Professor Ben Liblit for their insightful feedback and help. This work is supported in part by NSF grants CCF-1018180, CCF-1054616, and CCF-1217582; and a Clare Boothe Luce faculty fellowship.

References

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

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[3] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, 2010.

[4] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *ASE*, 2011.

[5] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In *Proceedings of the 18th European conference on Machine Learning*, 2007.

[6] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run software failure diagnosis via hardware performance counters. In *ASPLOS*, 2013.

[7] M. Attariyan, M. Chow, and J. Flinn. X-ray: automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[8] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=944919.944937`.

[9] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[10] A. Diwan, M. Hauswirth, T. Mytkowicz, and P. F. Sweeney. Traceanalyzer: a system for processing performance traces. *Softw., Pract. Exper.*, 41(3):267–282, 2011.

[11] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *FSE*, 2008.

[12] R. Fonseca, M. J. Freedman, and G. Porter. Experiences with tracing causality in networked services. In *Internet network management conference on Research on enterprise networking*, 2010.

[13] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. C. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *SOSP*, 2009.

[14] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE*, 2005.

[15] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, 2012.

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[17] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, 2010.

[18] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, 2012.

[19] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.

[20] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *FAST*, 2010.

[21] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *FSE*, 2010.

[22] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[23] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[24] T. Liu and E. D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *OOPSLA*, 2011.

[25] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, 2014.

[26] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[28] I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.

[29] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *The 10th Working Conference on Mining Software Repositories*, 2013.

[30] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, 2013.

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

[32] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, 2011.

[33] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, 2009.

[34] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance*, 2000.

[35] X. Xiao, S. Han, T. Xie, and D. Zhang. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA*, 2013.

[36] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, 2010.

[37] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI*, 2009.

[38] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, 2010.

[39] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.

[40] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ASPLOS*, 2014.

[41] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *The 9th Working Conference on Mining Software Repositories*, 2012.

[42] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.