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 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 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.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification – statistical methods; D.2.5 [Software Engineering]: Testing and Debugging – debugging aids

General Terms Languages, Measurement, Performance, Reliability

Keywords empirical study; performance diagnosis; performance bugs; statistical debugging

1. Introduction
1.1 Motivation
Implementation or design defects in software can lead to inefficient computation, causing unnecessary performance losses at run time. Previous studies have shown that this type of performance-related software defects widely exist in real-world [10, 21, 24, 35, 41]. They are difficult for developers to avoid due to the lack of performance documentation of APIs and the quickly changing workload of modern software [21]. A lot of performance bugs escape the in-house testing and manifest during production runs, causing severe performance degradation and huge energy waste in the field [21]. Making things worse, the negative impact of these performance problems is getting increasingly important, 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.

The state of practice of performance diagnosis is preliminary. The most commonly used and often the only available tool during diagnosis is profiler [1, 38]. 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.

* Shan is now with University of Chicago.

1 We will refer to these defects as performance bugs or performance problems interchangeably following previous work in this area [21, 24, 37].
2 In this paper, root cause refers to a static code region that can cause inefficient execution.
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 [37], under-utilized data structures [46], and temporary object bloat [12] [44] [45], 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. Many useful and generic techniques [16] [19] [20] [22] [29] [50] have been proposed. Among these techniques, statistical debugging is one of the most effective [20] [22] [29]. 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 [50], 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 [29] [30]; interleaving-related predicates have been designed to diagnose concurrency bugs [6] [20]; ∆LDA statistical model [5] has been used to locate failure root causes that have weak signals. What type of predicates and statistical models, if

```c
void startBulkInsert(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 1: A real-world performance bug in MySQL (the '-' and '+' demonstrate the patch)
any, would work well for performance diagnosis is still an open question.

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 understand whether it is feasible to apply statistical 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 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 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 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 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.

2.1 Methodology

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

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 [21]. We briefly discuss this baseline benchmark suite and our refinement below.

The baseline benchmarks [21] 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.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Apache</th>
<th>Chrome</th>
<th>GCC</th>
<th>Mozilla</th>
<th>MySQL</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparison within one code base</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparing the same input with different configurations</td>
<td>9</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>12</td>
<td>38</td>
</tr>
<tr>
<td>Comparing inputs with different sizes</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Comparing inputs with slightly different functionality</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td><strong>Comparison cross multiple code bases</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparing the same input under application’s different versions</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>Comparing the same input under different applications</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td><strong>Not using comparison-based methods</strong></td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

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)

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, 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 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 hasn’t” 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
Table 3: Inputs provided in users’ bug reports ($n$: developers provide a way to generate a large number of inputs)

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

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 froze 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.

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 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 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 (50).

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 by previous empirical studies, the time to diagnose user-reported performance problems is slightly shorter than that for non-user-reported performance problems [21], and similar or longer than that of functional bugs [21][33].

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. 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 [28]. 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 [42] show that the above observation is statistically 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 [42] 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 [42] 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 [50]. This task is likely easier for performance problem diagnosis.

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, 20, 22, 29, 30, 40] 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.

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 $P$ 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 [29, 30].

Returns. There are a set of six return predicates for each function return point, tracking whether the return value is ever $< 0$, $\leq 0$, $> 0$, $\geq 0$, $= 0$, or $\neq 0$ [29, 30].

Scalar-pairs. There are six scalar-pair predicates for each pair of variables $x$ and $y$, tracking whether $x$ is ever $< y$, $\leq y$, $> y$, $\geq y$, $= y$, or $\neq y$. 
$\geq y$, $\geq y = y$, or $\neq y$. 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, 22, 40].

**Interleaving-related ones.** Previous work on diagnosing concurrency bugs [20] 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**

Many statistical models have been used before for anomaly detection [13, 18, 26, 27] and fault localization [4, 5, 20, 22, 29, 30, 40]. 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 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 [6, 20, 29, 30] 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 [29, 30] that belongs to the first class discussed above and a LDA model proposed by Andrzejewski et al. [5] 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 [5, 29, 30], we briefly discuss these two models below. More details about these two models can be found in their original papers [5, 29, 30].

**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 [29].

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

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

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

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

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

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 [30] 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.

**LDA model** LDA [5] 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 [5]. Imagine the following scenario — during a success run, predicate $P$ is true for 10 times and false for 10 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, LDA 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 LDA model is very complicated, and is skipped here. It can be found in previous work [5].

**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
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. *: we have no tools to collect scalar-pair predicates in C++ programs. The 1s and n's in the “Reported Inputs” column indicate how many bad/good inputs are reported by users.)

<table>
<thead>
<tr>
<th>BugID</th>
<th>KLOC</th>
<th>Language</th>
<th>Static # of predicates</th>
<th>Static # of Reported Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Branch</td>
<td>Return</td>
</tr>
<tr>
<td>Mozilla258793</td>
<td>3482</td>
<td>C++</td>
<td>385722</td>
<td>1126770</td>
</tr>
<tr>
<td>Mozilla299742</td>
<td>3482</td>
<td>C++</td>
<td>385720</td>
<td>1126698</td>
</tr>
<tr>
<td>Mozilla347306</td>
<td>88</td>
<td>C</td>
<td>26804</td>
<td>38634</td>
</tr>
<tr>
<td>Mozilla416628</td>
<td>105</td>
<td>C</td>
<td>28788</td>
<td>39306</td>
</tr>
<tr>
<td>MySQL15811</td>
<td>1149</td>
<td>C++</td>
<td>13508</td>
<td>15576</td>
</tr>
<tr>
<td>MySQL26527</td>
<td>986</td>
<td>C++</td>
<td>90128</td>
<td>128610</td>
</tr>
<tr>
<td>MySQL27287</td>
<td>995</td>
<td>C++</td>
<td>92316</td>
<td>119322</td>
</tr>
<tr>
<td>MySQL40337</td>
<td>1191</td>
<td>C++</td>
<td>103686</td>
<td>138582</td>
</tr>
<tr>
<td>MySQL42649</td>
<td>1164</td>
<td>C++</td>
<td>126822</td>
<td>155766</td>
</tr>
<tr>
<td>MySQL44723</td>
<td>1164</td>
<td>C++</td>
<td>126822</td>
<td>155766</td>
</tr>
<tr>
<td>Apache3278</td>
<td>N/A</td>
<td>Java</td>
<td>10</td>
<td>126</td>
</tr>
<tr>
<td>Apache34464</td>
<td>N/A</td>
<td>Java</td>
<td>22</td>
<td>42</td>
</tr>
<tr>
<td>Apache47223</td>
<td>N/A</td>
<td>Java</td>
<td>24</td>
<td>36</td>
</tr>
<tr>
<td>Apache32546</td>
<td>N/A</td>
<td>Java</td>
<td>6</td>
<td>66</td>
</tr>
<tr>
<td>GCC1687</td>
<td>2099</td>
<td>C</td>
<td>183496</td>
<td>296058</td>
</tr>
<tr>
<td>GCC8805</td>
<td>2538</td>
<td>C</td>
<td>207188</td>
<td>327804</td>
</tr>
<tr>
<td>GCC15209</td>
<td>2586</td>
<td>C</td>
<td>192108</td>
<td>304800</td>
</tr>
<tr>
<td>GCC21430</td>
<td>3844</td>
<td>C</td>
<td>238514</td>
<td>447510</td>
</tr>
<tr>
<td>GCC46401</td>
<td>5521</td>
<td>C</td>
<td>337810</td>
<td>713532</td>
</tr>
<tr>
<td>GCC12322</td>
<td>2341</td>
<td>C</td>
<td>177098</td>
<td>284484</td>
</tr>
</tbody>
</table>

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. 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 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 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, ∆LDA) for statistical debugging. For C programs, we use CBI [29, 30] to collect all these three types of predicates. For C++ programs, we implement our own branch-predicate and return-predicate collection tools using PIN binary-instrumentation framework [34]. 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). 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 ∆LDA 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 ∆LDA model is the number of bad topics. We use the default setting 1.

We also use OProfile [38] 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. “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 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.

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,

```c
-cond
-cond
-cond
-cond
-cond
-cond
-timedwait
-timedwait

ware failures.

failure runs, and utilize statistical model to identify the likely causes of soft-
tical debugging. It collects predicate information from both success and
failure runs, and utilize statistical model to identify the likely causes of soft-
ware failures.

CBI [29, 30] is a C framework for lightweight instrumentation and sta-
tistical debugging. It collects predicate information from both success and
failure runs, and utilize statistical model to identify the likely causes of soft-
ware failures.

4

number one by the statistical model is indeed highly related
more, in all these 8 cases, the failure predictor that is ranked

∆LDA model for statistical debugging. For C

LDA model is the number of bad topics. We
use the default setting 1.

We also use OProfile [38] 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. “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 provided by OProfile for each
function in its ranking list. It first checks whether any di-
rect 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.

3.2.2 Results for basic model
Overall, 8 out of 20 performance problems can be success-
fully 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,

```
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 4 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 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 mysqli, and the patch makes mysqli 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 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. That is, for MySQL44723, 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

<table>
<thead>
<tr>
<th>BugID</th>
<th># of candidate predicates</th>
<th>Basic model</th>
<th>ALDA model</th>
<th>Profiler</th>
<th>Developers’ fix strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Branch</td>
<td>Return</td>
<td>S-pair</td>
<td>Branch</td>
<td>Return</td>
</tr>
<tr>
<td>Mozilla258793</td>
<td>62822</td>
<td>149354</td>
<td>*</td>
<td>✓ (0)</td>
<td>-</td>
</tr>
<tr>
<td>Mozilla299742</td>
<td>61256</td>
<td>148689</td>
<td>*</td>
<td>✓ (0)</td>
<td>-</td>
</tr>
<tr>
<td>Mozilla347306</td>
<td>3931</td>
<td>4062</td>
<td>21590</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Mozilla416628</td>
<td>3719</td>
<td>3598</td>
<td>19428</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MySQL15811</td>
<td>1198</td>
<td>866</td>
<td>*</td>
<td>✓ (0)</td>
<td>-</td>
</tr>
<tr>
<td>MySQL26527</td>
<td>62822</td>
<td>149354</td>
<td>*</td>
<td>✓ (0)</td>
<td>-</td>
</tr>
<tr>
<td>MySQL27287</td>
<td>5377</td>
<td>5752</td>
<td>*</td>
<td>✓ (0)</td>
<td>-</td>
</tr>
<tr>
<td>MySQL40337</td>
<td>7868</td>
<td>8160</td>
<td>✓ (1)</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>MySQL42649</td>
<td>12569</td>
<td>9696</td>
<td>✓ (1)</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>MySQL44723</td>
<td>10476</td>
<td>9108</td>
<td>✓ (1)</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>Apache3278</td>
<td>7</td>
<td>63</td>
<td>102</td>
<td>✓ (1)</td>
<td>✓ (2)</td>
</tr>
<tr>
<td>Apache34464</td>
<td>17</td>
<td>23</td>
<td>193</td>
<td>✓ (0)</td>
<td>✓ (2)</td>
</tr>
<tr>
<td>Apache47223</td>
<td>17</td>
<td>15</td>
<td>237</td>
<td>✓ (1)</td>
<td>✓ (1)</td>
</tr>
<tr>
<td>Apache32546</td>
<td>5</td>
<td>34</td>
<td>69</td>
<td>✓ (1)</td>
<td>✓ (1)</td>
</tr>
<tr>
<td>GCC1687</td>
<td>22602</td>
<td>17787</td>
<td>428103</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GCC8805</td>
<td>23891</td>
<td>20467</td>
<td>404594</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GCC15209</td>
<td>8956</td>
<td>9403</td>
<td>155007</td>
<td>✓ (13)</td>
<td>✓ (13)</td>
</tr>
<tr>
<td>GCC21430</td>
<td>45494</td>
<td>51270</td>
<td>647228</td>
<td>✓ (0)</td>
<td>✓ (0)</td>
</tr>
<tr>
<td>GCC446401</td>
<td>34365</td>
<td>38263</td>
<td>479508</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GCC12322</td>
<td>46721</td>
<td>8269</td>
<td>878823</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5: Experimental results for in-house diagnosis (✓, (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. ✓ (y): a y-th level caller of the x-th ranked function in a profiler result is related to the root cause; ✓ (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.)
Table 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)

<table>
<thead>
<tr>
<th></th>
<th>Apache</th>
<th>Chrome</th>
<th>GCC</th>
<th>Mozilla</th>
<th>MySQL</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total # of bugs</td>
<td>16</td>
<td>5</td>
<td>9</td>
<td>19</td>
<td>16</td>
<td>65</td>
</tr>
<tr>
<td># of bugs the default CBI model can help</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>Returns</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Scalar-Pairs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># of bugs ∆LDA model can help</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branches&lt;sub&gt;loop&lt;/sub&gt;</td>
<td>10</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>10</td>
<td>43</td>
</tr>
<tr>
<td>Returns</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Scalar-Pairs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># of bugs above designs cannot help</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

is the best failure predictor, the fix affects the return value of the corresponding function.

3.2.3 Results for ∆LDA model

We also tried statistical debugging using the ∆LDA 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 “Branch<sub>loop</sub>” in Table[5].

As shown in Table[5], ∆LDA 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 ∆LDA model.

Among the three different types of predicates, branch predicates are the most useful — help diagnosing 11 cases under ∆LDA model. In general, when a loop-branch predicate is considered as a failure predictor by the ∆LDA 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. Developers fix this problem by (1) completely removing the inefficient loop from the program (indicated by “Remove the loop” in Table[5]); (2) reduce the workload of the loop (indicated by “Reduce # loop iterations” in Table[5]); or (3) remove redundancy across loop iterations or across loop instances (indicated by “Combine loop iterations” or “Combine loop instances” 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 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 ∆LDA 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 ∆LDA model, but their diagnosis capability is subsumed by branch<sub>loop</sub> predicates in our evaluation, as shown in Table[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 branch<sub>loop</sub> 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 branch<sub>loop</sub> failure predictor.

Comparing with the profiler ∆LDA 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, ∆LDA 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 ∆LDA 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 ∆LDA 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.
<table>
<thead>
<tr>
<th>Fix Categories</th>
<th>Apache</th>
<th>Chrome</th>
<th>GCC</th>
<th>Mozilla</th>
<th>MySQL</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total # of loop-related bugs</td>
<td>10</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>10</td>
<td>43</td>
</tr>
<tr>
<td>Remove the loop</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Combine loop instances (removing cross-loop redundancies)</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Reduce # loop iterations (reduce the workload of the loop)</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Combine loop iterations (removing cross-iteration redundancies)</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Others</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 7: Fix strategies for loop-related bugs

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) 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.

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 6.

Assuming the basic statistical model, traditional predicates (i.e., branches, returns, and scalar-pairs) can diagnose 15 out of 65 performance problems. Among them, branch predicates are the most helpful, able to diagnose 13 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, 43 of them are caused by inefficient loops. We expect that the ΔLDA statistical model can identify root-cause related branch predicates (denoted as “Branches\textsubscript{loop}” in Table 3). 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 6.

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, 20, 22, 29, 30, 40] 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.

To guide future research on performance diagnosis, we further studied those 43 loop-related performance problems, and manually categorized their fix strategies, as shown in Table 7. 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 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 run-time 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.

4.1 Design

The state of the art on production-run functional bug diagnosis [6, 20, 29, 30] 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 [6, 20, 29, 30]. 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 diagnosis [6, 20, 29, 30], 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 [29, 30] that can discover multiple failure predictors corresponding to different root causes mixed in one profile pool, as well as some failure bucketing techniques [15] 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 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.

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** To sample return predicates, we directly use CBI [29, 30]. 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 [6]. 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) run-time 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 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 run-time 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, Section 3. That is, basic model plus branch predicates are used for seven benchmarks; basic model plus return predicates are used for one benchmark; ALDA model plus branch loop predicates are used for the remaining twelve benchmarks, including GCC12322. Since sampling can only lower overhead and cannot improve the diagnosis capability, those combinations that fail to deliver useful diagnosis results in Table 5 still fail to deliver useful diagnosis results in our sampling-based evaluation.

4.2.2 Results

Run-time overhead. As shown in Table 8 the run-time 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 8 it can be further lowered to be mostly below 2% under the $\frac{1}{100}$ sampling rate, and could be as large as over 40% under the $\frac{1}{1000}$ sampling rate.

Diagnosis capability. As shown by Table 8 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 benchmarks, the rankings of the ideal failure predictors remain the same as those without sampling in Table 5.

Also as expected, the diagnosis capability would decrease under sparser sampling or fewer failure/success runs. As shown in Table 9 under the default setting of 1000 success/failure runs, the diagnosis capability is roughly the same between $\frac{1}{10}$ sampling rate and $\frac{1}{100}$ sampling rate, but would drop with $\frac{1}{1000}$ sampling rate. Four benchmarks that can be diagnosed with more frequent sampling cannot be diagnosed with $\frac{1}{1000}$ sampling rate. Clearly, more runs will be needed to restore the diagnosis capability with a lower sampling rate.

Diagnosis latency. Diagnosis latency versus run-time overhead and diagnosis capability is a fundamental trade-off facing sampling-based statistical debugging for functional bugs [20, 29, 30]. 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 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 3, where only 10 failure runs are used.

<table>
<thead>
<tr>
<th>BugID</th>
<th>Diagnosis Capability</th>
<th>Overhead (# of runs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(10)</td>
<td>(100)</td>
</tr>
<tr>
<td>Mozilla258793</td>
<td>-</td>
<td>✓1</td>
</tr>
<tr>
<td>Mozilla299742</td>
<td>-</td>
<td>✓1</td>
</tr>
<tr>
<td>Mozilla347306</td>
<td>✓1</td>
<td>✓1</td>
</tr>
<tr>
<td>Mozilla416628</td>
<td>✓1</td>
<td>✓1</td>
</tr>
<tr>
<td>MySQL15811</td>
<td>✓1</td>
<td>✓1</td>
</tr>
<tr>
<td>MySQL26527</td>
<td>-</td>
<td>✓1</td>
</tr>
<tr>
<td>MySQL27287</td>
<td>✓1</td>
<td>✓1</td>
</tr>
<tr>
<td>MySQL40337</td>
<td>✓1</td>
<td>✓1</td>
</tr>
<tr>
<td>MySQL42649</td>
<td>-</td>
<td>✓2</td>
</tr>
<tr>
<td>MySQL44723</td>
<td>✓1</td>
<td>✓1</td>
</tr>
<tr>
<td>Apache3278</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Apache34464</td>
<td>✓3</td>
<td>✓3</td>
</tr>
<tr>
<td>Apache47223</td>
<td>✓1</td>
<td>✓1</td>
</tr>
<tr>
<td>Apache32546</td>
<td>✓1</td>
<td>✓1</td>
</tr>
<tr>
<td>GCC1687</td>
<td>✓1</td>
<td>✓1</td>
</tr>
<tr>
<td>GCC8805</td>
<td>✓4</td>
<td>✓4</td>
</tr>
<tr>
<td>GCC15209</td>
<td>-</td>
<td>✓1</td>
</tr>
<tr>
<td>GCC21430</td>
<td>✓1</td>
<td>✓1</td>
</tr>
<tr>
<td>GCC46401</td>
<td>✓2</td>
<td>✓2</td>
</tr>
<tr>
<td>GCC12322</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 8: Run-time 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.
Interestingly, there are 11 benchmarks, whose diagnosis latency is not lengthened by sampling. As shown in Table 8 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. 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 run-time 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, 1/10 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 1/10 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. Related Work

#### 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 compare the qualitative difference between performance bugs and non-performance bugs across impact, context, fix and fix validation; some of them look at how performance bugs are introduced, how performance bugs manifest, and how performance bugs are fixed; some of them 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. 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. IntroPerf [25] automatically infers the latency of user-level and kernel-level function calls based on operating system tracers. StackMine [17] automatically identifies certain call-stack patterns that are correlated with performance problems of event handlers. Yu et al. [43] 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 have been proposed to diagnose performance problems in distributed systems [2] [14] [23] [39] [47]. 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 aim to find a specific type of hidden performance bugs before the bugs lead to performance problems observed by end users.

Some tools [12] [44] [45] detect runtime bloat, a common performance problem in object-oriented applications. Xu et al. [46] targets low-utility data structures with unbalanced costs and benefits. Jin et al. [21] employ rule-based methods to detect performance bugs that violate efficiency rules that have been violated before. Chen et al. [9] detect database related performance anti-patterns, like fetching excessive data from database and issuing queries that could have been aggregated. WAIT [3] focuses on bugs that block the application from making progress. Liu and Berger [51] build two tools to attack the false sharing problem in multi-threaded software. There are also tools that detect inefficient nested loops [37] and workload-dependent loops [43].

These bug-detection tools have different focus from our work. They do not focus on diagnosing general performance problems reported by end users. Most of them are also not guided by performance symptoms.

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 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.

Acknowledgments

We thank the anonymous reviewers for their valuable comments which have substantially improved the content and presentation of this paper; Professor Darko Marinov and Professor Ben Liblit for their insightful feedback and help; Jie Liu for his tremendous help in statistical concepts and methods. This work is supported in part by NSF grants CCF-1018180, CCF-1054616, and CCF-1217582; and a Clare Boothe Luce faculty fellowship.

References


