The Journal of Systems and Software 156 (2019) 268–282

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Reproducing performance bug reports in server applications: The researchers' experiences

Xue Han, Daniel Carroll, Tingting Yu*

Department of Computer Science, University of Kentucky, Lexington, KY 40508, United States

ARTICLE INFO

Article history:
Received 22 October 2018
Revised 11 June 2019
Accepted 28 June 2019
Available online 29 June 2019

Keywords:
Performance bug reproduction
Bug characteristics study
Experience report

ABSTRACT

Performance is one of the key aspects of non-functional qualities as performance bugs can cause significant performance degradation and lead to poor user experiences. While bug reports are intended to help developers to understand and fix bugs, they are also extensively used by researchers for finding benchmarks to evaluate their testing and debugging approaches. Although researchers spend a considerable amount of time and effort in finding usable performance bugs from bug repositories, they often get only a few. Reproducing performance bugs is difficult even for performance bugs that are confirmed by developers with domain knowledge. The amount of information disclosed in a bug report may not always be sufficient to reproduce the performance bug for researchers, and thus hinders the usability of bug repository as the resource for finding benchmarks. In this paper, we study the characteristics of confirmed performance bugs by reproducing them using only informations available from the bug report to examine the challenges of bug reproduction from the perspective of researchers. We spent more than 800 h over the course of six months to study and to try to reproduce 93 confirmed performance bugs, which are randomly sampled from two large-scale open-source server applications. We (1) studied the characteristics of the reproduced performance bug reports; (2) summarized the causes of failed-to-reproduce performance bug reports from the perspective of researchers by reproducing bugs that have been solved in bug reports; (3) shared our experience on suggesting workarounds to improve the bug reproduction success rate; (4) delivered a virtual machine image that contains a set of 17 ready-to-execute performance bug benchmarks. The findings of our study provide guidance and a set of suggestions to help researchers to understand, evaluate, and successfully replicate performance bugs.

1. Introduction

Software performance is critical to the quality of the software system. Unlike functional bugs that typically cause system crashes or incorrect results, a performance bug can cause significant performance degradation (Attariyan et al., 2012) which leads to problems such as poor user experience, long response time, and low system throughput (Bugzilla, 2016; Han et al., 2012; Jin et al., 2012; Nistor et al., 2013a; Wert et al., 2013). For instance, performance bugs have occurred on well-tested software such as the Internet Explorer installed on Windows systems (Han et al., 2012), and have caused severe damages to the user experience.

Compared to functional bugs, performance bugs are substantially more difficult to handle (Attariyan et al., 2012; Dean et al., 2014) because they often manifest themselves through large inputs and specific execution environments (Nistor et al., 2013a; Olivo et al., 2015). Thus, traditional testing such as coverage-based approaches may not be effective. To address performance issues, numerous research efforts, especially on dynamic techniques, have been made to analyze, detect, and fix performance bugs (Burnim et al., 2009; Han et al., 2012; Jin et al., 2012; Jovic et al., 2011; Nistor et al., 2013b; Olivo et al., 2015). Although these techniques can detect performance bugs in the benchmark applications they studied, their effectiveness in real-world large-scale software projects, such as server applications, is largely unknown. This is partly due to the fact that finding performance bugs to be used for evaluation is difficult.

Many modern software projects use bug tracking systems (e.g., Bugzilla, 2016, Github Issue Tracker (GitHub, 2008)) that allow developers and users to report issues they have identified in the software. While bug reports are intended to help developers to understand and fix bugs, they are also used by researchers to evaluate a proposed testing or debugging approach. Researchers often rely on the description of a confirmed performance bug

* Corresponding author.
E-mail addresses: xue.han@uky.edu (X. Han), drca234@g.uky.edu (D. Carroll), tyu@cs.uky.edu (T. Yu).

https://doi.org/10.1016/j.jss.2019.06.100
0164-1212/© 2019 Elsevier Inc. All rights reserved.

report to reproduce the performance bug to be used in their evaluation. A failed-to-reproduce performance bug in this work is defined as a confirmed reproducible performance bug that cannot be reproduced by non-domain experts such as researchers due to the lack of domain knowledge or environment limitations (e.g., compilation, dependencies, etc). A failed-to-reproduce performance bug is likely to be discarded by researchers when it cannot be reproduced according to the bug report. Therefore, the bug selection and reproduction process is very challenging and may discourage researchers from trying a lot of potential bugs that are of the interest to the proposed approach.

In a recent paper Dean et al. (2014) on dynamic detection of performance bugs, the authors state "the bug reproduction is extremely time-consuming and tricky due to limited and often ambiguous information, which sometimes takes a month for us to reproduce one bug". In more than 30 performance testing and diagnosis papers we studied, none of them described how performance bugs are reproduced. To the best of our knowledge, there is no study or experience report showing what has caused performance bugs to be so difficult to understand and reproduce.

A high-quality bug report requires inputs, reproducing steps, and test oracles. One challenge for performance testing tools is that they generally require a large amount of workload or specific environment settings to expose performance bugs. However, in our experience, we found that even using the described inputs, reproducing steps, and test oracles from bug reports, performance bugs may still not be reproduced. One natural question to ask is what would be the other factors that lead to failed-to-reproduce bugs beyond the quality of bug report itself. It would be very helpful if we can identify these factors, and suggest solutions to the failed-to-reproduce bugs to increase the chance of success in performance bug reports reproduction.

The goal of this work is to share our experience in reproducing performance bug reports by investigating the impact of different factors on both reproduced and failed-to-reproduce performance bugs from open-source project confirmed performance bug reports. We provide a set of workarounds to increase the chance of success in performance bug reproduction. Our study targets reproducing performance bugs from the perspectives of non-domain expert researchers, rather than understanding and characterizing non-reproducible bugs from the viewpoints of developers. Therefore, the scope of our study focuses on performance bug reports that have already been confirmed and resolved by developers. One big difference compared to the prior work is that we specifically target confirmed performance bugs to report why from the perspective of non-domain experts such as software engineering researchers may not succeed in reproducing such bugs. We studied two large open-source server projects: Apache HTTP Server and MySQL database. Because performance bugs are more prevalent in applications that are large-scale and handle a large quantity of data over a long period of time, we focus on server applications. We randomly selected, analyzed, and conducted reproduction of 93 bugs in total. The results of this study mainly aim to help researchers to better understand the challenges in performance bug reproduction and propose solutions to facilitate the bug selection process.

The main findings and contributions of our study are as follows:

- We tried to reproduce performance bugs that were solved by developers by following the description of the bug reports. After six months of effort, we were able to reproduce 17 out of 93 bugs. We found that a majority of performance bugs (81%) failed to be reproduced.
- We studied the characteristics of 17 reproduced performance bug reports. A majority (88%) of them can be reproduced with no more than three inputs and most (53%) of them required specific workloads; 10 bug reports involved transient performance bugs that must be observed during the reproduction. A significant portion (59%) of reproduced performance bug reports required more than two action steps.
- Among 17 reproduced performance bugs, only two of them can be reproduced by directly following the bug report description. However, the other 15 bugs required workarounds to be reproduced.
- We studied different factors of performance bugs that we failed to reproduce after months of effort. These factors include hardware dependency, OS dependency, component dependency, source code unavailability, compilation error, installation error, missing step, and lack of symptoms. Missing step, OS dependency, and lack of symptoms were in the dominant majority (39%).
- We further examined reasons why performance bugs failed to be reproduced on the first attempt. We provided a list of strategies for increasing the chance of successfully reproducing the performance bugs.
- While this study primarily targets researchers in selecting performance bugs, we provided a set of implications for both researchers and practitioners on developing techniques for testing and diagnosing performance bugs, improving the quality of bug reports, and detecting failed-to-reproduce bug reports.
- We made our datasets publicly available and provided a virtual box image that contains 17 benchmark programs.[1]

The rest of the paper is organized as follows. We first present motivating examples in Section 2. We then describe our methodology for choosing subject applications, the bugs selected to study, and the threats to validity in Section 3. Our results are demonstrated in Section 4, followed by discussions in Section 5. We present related work in Section 6 and end with conclusions in Section 7.

2. Motivating examples

We refer to software performance bugs as programming (Nistor et al., 2013a; 2013b) and configuration errors that cause significant performance degradation. They can adversely affect speed, throughput, and responsiveness of the system, which lead to the poor user experience. Some other terms such as "performance problem" and "performance issue" are also widely used (Nistor et al., 2013a). In this paper, we use these terms interchangeably.

We use three examples of performance bug reports to answer the following questions: (1) What are limitations in confirmed bug reports that can lead to the failed-to-reproduce performance bugs? (2) What can we do to increase the chance to successfully reproduce a confirmed performance bug?

We illustrate three difficulty levels of reproducing confirmed performance bugs from the perspective of non-domain experts such as software engineering researchers with three bug reports. The difficulty level ranges from hard-level ("failed-to-reproduce") to medium-level ("reproducible with effort") to easy-level ("reproducible").

2.1. Hard-level: Apache bug #58037

The bug reporter observed a noticeable time delay in Lightweight Directory Access Protocol (LDAP) authentication after the Apache server was upgraded from version 2.2 to version 2.4. However, we were not able to reproduce this performance bug for several reasons. First, the minor version of the faulty Apache server

[1] https://github.com/xha225/PerfBugReplication.

was not mentioned in the bug report. Since there are 34 releases under Apache v2.2, on average, compiling and installing Apache from source code can take anywhere between 20 and 50 min; it is too time-consuming (up to 28 h in the worst-case) to pinpoint the faulty version. We finally adopted a version that is closest to the timeline when the performance bug was reported, but we were still not able to reproduce the bug because of the other two reasons.

Second, the bug report indicates that configuration option `LDAPConnectionPoolTTL` in the LDAP module must be set to 0 for reproducing the bug. Since exposing the bug heavily relies on the LDAP module, we believe that more than one configuration option must be set to proper values, but they are not mentioned in the report. Third, the bug report describes the symptom as "we noticed that it would take longer and longer to check out a large repository." It is not clear about how large "a large repository" is. However, such information is essential to closely resemble the required input loads to reproduce the performance bug and to observe the expected symptom. Although we used our best guesses to set up the program and the environment, tried different levels of input workloads, and followed the reproduction steps as closely as possible, we still failed to observe the symptom described in the bug report.

2.2. Medium-level: Apache Bug #27106

The bug reporter observed a memory leak that led to a system slowdown when running tests using the Apache benchmark. Specifically, when testing using an HTTP request with an SSL-enabled port, memory used by the `httpd` process grew rapidly. While the bug report did describe the bug-triggering inputs (i.e., HTTP request) and the observed symptom (increased memory usage), we were still having a lot of trouble reproducing the bug.

First, the description of the environment setup was ambiguous. The information of the Linux operating system (OS) version under which the bug happened was missing. In addition, dependency modules, such as the OpenSSL module, that should be enabled with the Apache server v2.0.45 are not mentioned. Apache must be re-configured to include the OpenSSL module during the compile time. Second, the description of inputs is incomplete. The bug reporter suggest using Apache benchmark (`ab`) to trigger the bug, but the parameters passed to the `ab` are not specified. Apache benchmark that comes with v2.0.45 does not support Hypertext Transfer Protocol Secure (HTTPS). We need to find an ab version that does support HTTPS. Third, the description of the observed symptoms was unclear. The bug reporter should have asked users to watch memory usage on the main thread of Apache (e.g. by using the Linux system monitoring tools such as `ps` to show process status). Instead, the reporter posted a raw trace and let readers figure out what information is important.

To reproduce this bug report, we spent about 10 h to research on plausible components to fill in the missing information to successfully reproduce the performance bug. We first build Apache with default settings to make sure the specific version (v2.0.45) works. We use the release date of Apache v2.0.45 to identify a compatible OpenSSL version (i.e., OpenSSL v0.9.7a). To observe the performance bug symptom, we use the Apache benchmark `ab` to request 10,000 pages with 50 threads enabled: "ab -n 10,000 -c 50 https://localhost:443/".

2.3. Easy-level: MySQL Bug #74325

This performance regression bug happens in MySQL v5.7.5. When compared to MySQL v5.7.5, MySQL v5.0.85 is four times faster in updating an indexed column. The bug reporter provides concrete information on the bug-triggering inputs, the environment setup, and the observed bug symptom.

First, the input passed to the `mysqlslap` benchmark tool is specified. The bug reporter also suggests that specific configuration options (e.g. `query_cache_size`) are needed for triggering the performance bug. Second, the description of the environment setup is accurate and concise. The reporter clearly indicates the MySQL version (i.e., v5.7.5) from which the performance deterioration can be observed, as well as the software components and their versions that MySQL v5.7.5 depends on. Finally, the description of symptom is clear enough to determine the performance bug: "InnoDB is more than 2X slower than5.6.21" in MySQL v5.7.5 "when updating to indexed column". Since this bug report contains more detailed information than the other two bugs, we spent about 5 h to successfully reproduce the bug.

3. Case study

Our study has two main objectives. First, we intend to understand why reproducing performance bugs from bug reports are challenging. Second, we want to understand how to design solutions to increase the chance of successfully reproducing performance bugs. Therefore, we consider the following research questions.

RQ1: How difficult is it to reproduce performance bug reports and what are the characteristics of the reproduced bug reports?

RQ2: What are the major factors that cause reproducing confirmed performance bug reports to fail?

RQ3: What strategies can be used to improve the chance of success in reproducing confirmed performance bug reports?

3.1. Data sets

3.1.1. Studied subjects

We chose two large popular open-source server projects: Apache HTTP Server and MySQL Server. With publicly accessible code base and well-maintained bug systems, these two subjects have been widely used by existing bug characteristic studies (Jin et al., 2012; Yin et al., 2011; Zaman et al., 2012). The selected programs are listed in Column 1 of Table 1. Both projects started in the early 2000s and each has over ten years of bug reports.

3.1.2. Data collection

We collected performance bugs from the bug system of Apache (ASFB, 2016) and MySQL (MySQLBug, 2016). We searched bug systems using a set of commonly used general keywords and phrases to describe the symptoms of performance bugs, such as "slow", "latency", and "low throughput". We also searched terms that attribute to a specific aspect of the performance problems such as "CPU spikes", "cache hit", and "memory leak" to identify performance bugs. Next, for Apache, we selected bug reports with a status field of "RESOLVED", "VERIFIED", or "ClOSED", and a resolution field of "FIXED". For MySQL, we selected bug reports marked as "FIXED" or "PATCH APPROVED/QUEUED", and with the severity level field set to "NOT FEATURE REQUEST". We focus on fixed/closed reports because when examining bug reports to find executable benchmarks, they are more reliable than open bug reports and often adopted by researchers for evaluation purposes (Burnim et al., 2009; Han et al., 2012; Jin et al., 2012; Jovic et al., 2011; Nistor et al., 2013b; Olivo et al., 2015). More importantly, the decision to choose bug reports from confirmed bug reports is in line with our study goal, that is, to explore and experience from the viewpoint of researchers, how challenging it is to try to reproduce performance bugs from bugs reports that are considered to be reproducible by dedicated application developers. Some but not all projects provide designated tags for different categories

Table 1
Subject characteristics.

Subject	Init Rel	Last Rel	#Sampled	#Failed	#Rep	Success Rate
Apache 2.0	2002	2013	20	16	4	20%
Apache 2.2	2005	2017	31	26	5	16%
Apache 2.4	2012	2017	4	3	1	25%
MySQL 5.0	2005	2012	19	16	3	15%
MySQL 5.1	2008	2013	15	12	3	20%
MySQL 5.5+	2010	2017	4	3	1	25%
SUM	–	–	93	76	17	–

Init Rel. = The year of the initial release. Last Rel = The year of the most recent release.
#Sampled = Number of performance bug reports sampled in our study.
#Rep = Number of reproduced performance bug reports.
#Failed = Number of failed-to-reproduce performance bug reports.
Success Rate = Percentage of reproduced performance bug reports.

of bugs. For example, in the MySQL bug system, the bug severity tag "S5 (Performance)" is used to mark performance bugs. In our approach, we want to make the process as general as possible, therefore, our method does not rely on the performance tags.

The whole process yielded a total of 564 bugs. With a confidence level of 95% and a confidence interval of 5, the calculated sample size is 229. We randomly selected 229 bugs out of the 564 bugs and conducted a manual examination. During the manual inspection, we follow those bug reports that have sufficient information in bug descriptions and discussions posted by commentators, and decide whether the inspected bug is a performance bug or not. Specifically, the sufficient information includes bug symptoms involving performance issues, such as system's slow down, from the discussion of the bug report. If we cannot find the symptom information from the bug report, we cannot determine if the bug is a performance bug. For example, some bug reports simply provide a stack trace. We also examine the number of configuration options that are discussed in the various modules involved in the sampled bug reports. For Apache and MySQL, we have identified a total of 610 and 1240 configuration options respectively.

To ensure the correctness of our results, the manual inspections were performed independently by two inspectors (the first two authors). They both have experiences in using servers such as Apache and MySQL. We hold two discussion session to define what we consider to be a performance bug and how to keep a record of our findings. Each inspector is given the same set of bug reports each week, they meet twice a week to compare and consolidate their findings. A bug report is selected only when both inspectors agree on the outcome of the manual inspection. For bugs where the results differ, the authors and the inspectors discussed to reach a consensus. As such, the examination yielded a total of 93 performance bugs. Column Subject and #Sampled of Table 1 list the release versions of the subjects and performance bugs selected from each version.

3.1.3. Study setup

We build the environment as a virtual disk image (VDI) on the VirtualBox (VirtualBox, 2016) for flexibility and portability. The VDI provides great portability in the sense that it can be loaded as a disk image wherever the VirtualBox is installed. This convenience offers the possibility to provide a ready-to-run image for researchers without having them go through a lot of environment and project setup. The exact configuration other than the operating system is capped only by the host machine. The host machine is running on Mac OS X with a dual-core 3 GHz Intel Core i7 CPU, 16GB of memory, and 512GB of hard drive. For the guest machine (VDI), we use the Ubuntu 14.04 LTS operating system with a single core 3 GHz Intel Core i7 CPU, 4GB of memory, and 120GB of hard drive. For performance bugs that require more resources, we conduct experiments on a machine equipped with a 6 core 2.66 GHz Intel CPU, 36GB memory, and 256GB hard drive. For each of the 93 bugs, we ask both inspectors to follow the description of a bug report to reproduce the bug. The bug reproduction process involves two general steps: environment setup and performance bug reproduction. The whole process took around 800 h in total.

A bug is marked as *reproduced* if it can reveal the same symptom as described in the bug report. If we fail to reproduce a performance bug, the bug is marked as *failed-to-reproduce*. There are three major reasons when we mark a bug as failed-to-reproduce: (1) if the failure is due to the lack of hardware environment; (2) if the bug report provides insufficient instructions on steps to reproduce the bug; (3) if we follow all the steps in the bug report but cannot observe the symptom. Note that when we talk about a bug that is failed-to-reproduce, we do not mean the bug is indeed non-reproducible. We reserve the use of the term "non-reproducible bug" to application developers — only they can determine what bugs should be marked as non-reproducible. A failed-to-reproduce bug, on the other hand, is a bug with unknown reproducibility from our (the researchers') perspective. For a confirmed bug, the bug reporter should know how to reproduce the bug, however, crucial information, such as steps to reproduce the bug, may have been left out from the discussion in the bug report. For example, an extensive discussion may have been carried out in a mail-list, or through private messages and conversations. Regardless, such information may not be present in the bug report. From the perspective of researchers who try to reproduce a bug directly following the instructions available in the bug report, such bugs are considered failed-to-reproduce.

3.1.3.1. Environment setup. Before executing the program against its bug-triggering inputs to reproduce the bug, we will need to setup the execution environment. Environment setup typically involves choosing the target operating system, build, deploy the faulty program, and configure various software and hardware dependencies. An indication of passing the environment step includes the availability of the faulty program version, its dependencies, a successful build (if needed), and a functional program. For example, we failed to install the database server in MySQL bug #15811 with an error message saying "recipe for target 'install' failed". We could not find a solution to fix this installation error. Since we failed to reproduce the performance bug in the environment setup step, this bug is marked as *failed-to-reproduce*. As a matter of fact, less than half (41 out of 93) bugs passed the environment setup step.

3.1.3.2. Performance bug reproduction. After the bug reproduction environment has been successfully setup, the next step is to actually reproduce the bug. If the symptom of the bug matches what is described in the report, it is considered *reproduced*; otherwise it is considered *failed-to-reproduce*. Specifically, we execute the program against the bug-triggering inputs, follow its reproduction steps, and observe the output described in the bug report. The bug-triggering inputs often come from three sources: user inputs, configurations, and environment parameters (e.g., network bandwidth, memory, etc). A user input is often associated with a user-entered input (e.g., a file) or an input action such as issuing a particular HTTP request method (e.g., GET or POST) to request a particular type of web page. One or more configuration options are also sometimes needed to trigger performance bugs. For example, in Apache bug #37680, the configuration option "Listen" is required. In addition, inputs coming from external environment can affect reproducing a bug because exposing performance bugs may require the system to reach a specific level of load (e.g., a web server with a high volume of network traffic).

During the bug reproduction process, if no concrete input values for the three input sources are specified, we use random values or the default values provided by the program. For example, in MySQL bug #27501, we start the database server with default settings as no specific runtime configurations were provided.

We next follow the steps of bug reproduction described in the bug report (e.g., Apache bug #54852 has a section describing reproduction steps). If there are no specific steps provided, we try different solutions based on our experience and expertise. For example, Apache bug #43081 does not give enough information on the nature of a "busy" machine. We infer that the server is busy serving long connected requests based on the context of the bug report.

Finally, to determine whether a reported performance bug is successfully reproduced, we need to compare the observed symptom (i.e., long execution time) with the symptom described in the bug report. Unlike functional bugs in which their outputs are deterministic (e.g., an error message), performance bugs often use non-functional measures such as response time, throughput, and utilization (e.g., memory, and CPU usage) (Molyneaux, 2009). The values of such performance measures usually depend on the execution environment, so it is likely that a measured symptom observed in our execution environment is different from that described in the bug report. To address this problem, we examine the performance difference between a previous non-faulty version (or the patched version) and a faulty version. If the difference is proportional to the difference described in the bug report, the bug reproduction is considered successful.

3.2. Threats to validity

The primary threat to the external validity for this study involves the representativeness of our subjects and bug reports. Other subjects may exhibit different behaviors. Our study examines two popular open-source server applications (i.e. one web server and one database server) written in C/C++ and the result may not be generalized to other types of software such as client-side applications like a web browser. Data recorded in bug tracking systems and code version histories can have a systematic bias relative to the full population of bug fixes (Bird et al., 2009) and can be incomplete or incorrect (Aranda and Venolia, 2009). However, we do reduce this threat to some extent by using popular open-source projects and bug systems for our study. The second source of potential threat involves the age of bug reports. Since the sampled bug reports span over ten years, the way of performance bugs being reported may change as time passes by. This may somewhat affect the methodological consistency. We have examined these bug reports and found that such changes are minimal. The third threat is related to the type of bugs studied. In this paper, we are focusing on performance bugs, findings in performance bugs may not necessarily confirm in other types of bugs in terms of reproducibility.

The primary threat to the internal validity involves the use of keyword search and manual inspection to identify performance bugs. To minimize the risk of incorrect results given by the manual inspection, bug reports were labeled (as performance bugs or not) independently by two inspectors. The recall of our approach is estimated to be 50%. This is mainly due to fact that the keyword often does not carry a lot of information in the context. For instance, when searching for performance bugs with the keyword "slow", the bug returned talks about a potential fix for the bug may cause the application to slow down. The bug itself is not a performance issue. To compute this recall, we randomly sampled 100 bugs and manually inspected each of them. We found six performance bugs, of which only three were found by the keyword search and manual inspection. Such an approach is also used by Nistor et al. (2013a). The risk of not analyzing all performance bugs cannot be fully eliminated. However, combining keyword search and manual inspection is an effective technique to identify bugs of a specific type from a large pool of generic bugs, which has been successfully used in prior studies (Jin et al., 2012; Nistor et al., 2013a; Yin et al., 2011).

Another source of potential threat involves the lack of system resources (e.g., operating systems and hardware) necessary for reproducing certain performance bugs. Because Windows and Mac systems are proprietary, getting the appropriate license and OS image poses a higher challenge. We managed to try a few OSs (e.g. Windows) used in the bug report but failed to reproduce the bugs requiring specific OS versions. For Unix-like systems, some software components are no longer available. A few compilation problems may be fixed and more bugs could potentially be reproduced if we have the knowledge on the specific version of the compiler used to compile the subject discussed in a bug report. Last, certain workaround proposed may not be suitable for some researchers. For instance, even though in our experience we can use the binary executable instead of compiling the project from source code, this method may not work for researchers who wish to work on source code (e.g. developing code analysis techniques).

The primary threat to construct validity involves the dataset and metrics used in the study. To mitigate this threat, we used bug reports from the bug systems of the two subjects, which are publicly available and generally well understood. We have also used well-known metrics in our data analysis such as the number of bugs, which is straightforward to compute.

4. Results

We now present our results for each of the three research questions.

4.1. RQ1: Reproduced bug reports and their characteristics

Column #Rep and #Failed of Table 1 list the number of reproduced and failed-to-reproduce bugs across different versions of the two subjects.

Finding 1: A majority (82%) of reported performance bugs fail to be reproduced. The rate to successfully reproduce a performance bug report is low.

We next provide further details about the characteristics of the reproduced bug reports, shown in Table 2. The characteristics of the failed-to-reproduce bug reports will be discussed in Section 4.2.

When reproducing a server performance bug, environment setup, data inputs, configuration options, and input actions are four essential elements. Fig. 1 shows an example of the use of the four elements in Apache bug #48024. *Environment setup* refers to the

Table 2
Reproduced bugs and their characteristics.

Sub	BugID	Set	Inp	Opt	Load	Act	Order	Duration	Workaround
Apache	54852	12	0	1	YES	4	YES	Transient	YES
Apache	52914	9	2	2	NO	3	YES	Permanent	NO
Apache	37680	6	1	2	NO	7	YES	Permanent	YES
Apache	22030	12	1	0	NO	2	YES	Permanent	YES
Apache	51714	11	1	0	YES	7	YES	Permanent	YES
Apache	43081	10	0	6	YES	3	YES	Transient	YES
Apache	48024	13	1	3	YES	3	YES	Transient	YES
Apache	46749	16	1	2	YES	3	YES	Permanent	YES
Apache	27106	19	1	1	YES	2	YES	Permanent	YES
Apache	38017	10	1	9	NO	3	YES	Permanent	YES
MySQL	21727	8	1	1	NO	2	YES	Transient	YES
MySQL	44723	8	1	1	YES	2	YES	Transient	YES
MySQL	74325	14	1	2	YES	2	YES	Transient	YES
MySQL	15653	15	1	1	NO	3	YES	Transient	YES
MySQL	26938	16	1	1	NO	2	YES	Permanent	NO
MySQL	54989	11	1	1	NO	3	YES	Permanent	YES
MySQL	54914	13	1	1	YES	2	YES	Transient	YES
Avg.	–	12	1	2	YES (53%)	3	YES (100%)	Permanent (53%)	YES (88%)

Set = Number of steps to setup environment for reproducing the reported bug.
Inp = Number of input parameters. Opt = Number of configuration options.
Load = Whether a specific workload is needed.
Act = Number of input actions needed for triggering the bug after environment setup.
Order = Whether a specific order of actions is needed.
Duration = The duration of a performance bug symptom.
Workaround = Whether reproducing the bug requires workarounds.

Environment Setup
```
1. export INSTALL=$PWD/apache-install/
2. ./configure –prefix=$INSTALL –enable-sed –enable-proxy
...
12. python -m SimpleHTTPServer & #Start backend server
13. ./bin/apachectl start #Start proxy server
```
Data Input
A static file with 1M+ characters on a single line
Configuration Options
```
Header unset Content-Length
SetOutputFilter Sed
ProxyPass / http://127.0.0.1:8000/
```
Input Actions
HTTP request: http://localhost:8000/a.1

Fig. 1. Reproducing Apache Bug #48024.

steps to install OS and set up specific application components (e.g. in Fig. 1, we have enabled the sed module for Apache) that are required to reproduce a performance bug. *Data input* refers to the user-supplied data (e.g. in Fig. 1 we have used a static file that contains a single line) that is used to trigger a performance bug. *Workload* is the amount of processing that the computer has been given to do in a given time (Techtarget, 2006). Workload describes the intensity of data inputs. In the above example, the size of the input file defines the workload for the sed filter. *Configuration options* (e.g. in Fig. 1 we have included the ProxyPass option) correspond to the customizable items in the configuration file. The Apache mod_headers module "provides directives to control and modify HTTP request and response headers". For instance, "Header unset Content-Length" removes the Content-Length header. However, in this paper, we do not consider the HTTP headers as configuration options. Input action refers to the logical steps to take after environment setup for the performance bug to manifest (e.g. in Fig. 1, the action includes issuing an HTTP request).

Column Set of Table 2 lists the number of steps required for setting up the performance bug reproduction environment. We define a step as a single operation that can be completed by a shell command. For instance, to compile the source code with GNU `make` command is treated as one step.

Finding 2: Among 17 reproduced bug reports, a majority (65%) of them require more than 10 steps to setup the reproduction environment.

The results suggest that the environment for reproducing a performance bug is complex. Fig. 1 shows part of the 13 steps of environment setup for reproducing Apache bug #48024.

Columns Inp of Table 2 lists the number of input parameters (e.g., files) needed for triggering the performance bug. The results indicate that 15 out of 17 reproduced performance bugs require input parameters and 14 bugs require only one parameter. The input parameters in all 15 bugs involve files. This is because many operations offered by the subject applications require input files to function. For instance, in the Apache HTTP Server, when a request command is issued, it is normally associated with a type of file that is being requested, such as the example in Fig. 1. Occasionally, the content of the file plays an important role in triggering the performance bug. For example, in Apache bug #51714, the Perl script used to trigger the bug contains code to generate a large HTTP range header. In other cases, such as an Apache server restart, no input parameters are needed.

Finding 3: A large portion (82.3%) of reproduced bug reports require specific input parameters. A majority (13 out of 14) of them require only one input parameter.

Column Opt of Table 2 lists the number of configuration options (specified both as in configuration files and command line arguments) that lead to the performance bug. These options need to be set to particular levels, whereas values of the other options do not influence the exposure of the bug (or reproducibility of the bug) and thus can be set to arbitrary values. For example, in Apache bug #52914, two configuration options in the mod_reqtimeout module `RequestReadTimeout body` and `RequestReadTimeout header` are required to trigger a CPU spike. The results indicate

Table 3
Workload type.

Workload type	Description	Bug example
Web	Concurrent web page requests	Apache 54852
Traffic	Long HTTP connection sessions	Apache 43081
Database	Large number of database tables	MySQL 15653
	Concurrent updates on DB tables	MySQL 74325

that 15 out of 17 reproduced performance bug reports are related to specific configuration options. Such configuration options would require a value that goes above or below a threshold to trigger the performance bug, while other configuration option values remain default. 14 performance bugs require less than 3 configuration options.

Finding 4: A significant percentage (88.2%) of performance bugs require setting up specific configuration options to be reproduced. The majority (80%) of these bugs are related to only one option.

Column Load of Table 2 reports whether a specific level of workload is required for reproducing the bug. The results indicate that among 17 reproduced bugs, a majority (53%) of them need a specific level of workloads to trigger the bugs. For instance, in Apache bug #51714, a Perl script is used to generate a large volume of HTTP request loads. Each HTTP request header has a large value in the `Range` field to get bytes from the server. Table 3 summarizes the types of workloads in the 17 reproduced bug reports, including network traffic and database operations.

Finding 5: Almost half (53%) of the reproduced performance bugs require a specific level of workloads to manifest.

Column Act of Table 2 lists the number of input actions required for reproducing the reported performance bug. We define an input action as one logical step towards triggering the performance bug after the environment setup. For example, in Fig. 1, sending an HTTP request is an action.

Finding 6: A majority (88%) of the reproduced performance bug reports require no more than 3 input actions.

Column Order of Table 2 reports whether reproducing a reported performance bug requires a specific order of input actions. The results indicate that all 17 reproduced bug reports require multiple input actions to trigger the performance bugs. This is because our studied subjects are server programs, their reproductions must start with the action of *starting the server*. In MySQL bug #26938, a performance bug occurs as the database server froze over a list of recently used statements. To trigger this bug, the following steps are involved: (1) start a database server using "./bin/mysqld_safe"; (2) connect to a database server from a SQL client using "./bin/mysql"; (3) issue a SQL command using "show profile;". Nevertheless, the order of input actions matters in 9 bugs even after the server started. For example, to trigger a CPU spike in Apache bug #37680, a sequence of input actions must follow the specific order, as shown in Fig. 2.

```
Replace default port with "Listen 50000"
Add another port option "Listen 50001"
Restart sever
Make a request on port 50000
Delete option "Listen 50001"
Restart server
Make a request on port 50000
```

Fig. 2. Order of input actions.

Finding 7: The specific order of events is important in 52.9% of the reproduced bugs that require multiple input actions.

Column Duration of Table 2 reports the life span of the performance bug symptom. Permanent symptom indicates that the symptom is always observable once it is exposed, whereas transient symptom means that the symptom appears for a short period of time and then disappears.

Finding 8: A significant portion (47%) of bug reports involve transient symptoms.

For instance, in Apache bug #48024, when (1) Apache is configured as a reverse proxy server, (2) the SED respond content filter is enabled, and (3) a request to a file contains long characters in a single line, CPU suddenly spikes to 100%. However, this symptom is only observable when Apache is processing the requested file for about 5 s. Afterward, the CPU usage level returns to a normal state.

Column Workaround of Table 2 reports whether reproducing a performance bug requires efforts to workaround the difficulties (e.g., ambiguous description, version inconsistencies) in the report description.

Finding 9: A majority (88%) of reproduced bug reports require workarounds.

For example, in MySQL bug #44723, the `do_abi_check` block in Makefile.in fails the build due to a change of behavior in the later versions of GCC. After removing the block, MySQL compiles with no problems.

4.2. RQ2: Factors leading to failed-to-reproduce performance bug reports

Before we can improve the practice to increase the chance of success in reproducing performance bug reports, we want to identify major factors that cause the reproduction to fail. We classify the root causes of reproduction failures of the 76 failed-to-reproduce bugs into eight categories: hardware dependency, operating system (OS) dependency, component dependency, unavailable source code, compilation error, installation error, missing step, and lack of symptom. The eight categories are mutually exclusive when assigning bugs to a category. For instance, a bug report may have "missing step" but if we run into the "compilation error" problem, the bug report will not be counted under "missing step" unless we can workaround the "compilation error" step. In this case, the same bug will be counted once in each of the two categories. The distribution of performance bug reports in the eight categories is summarized in Fig. 3.

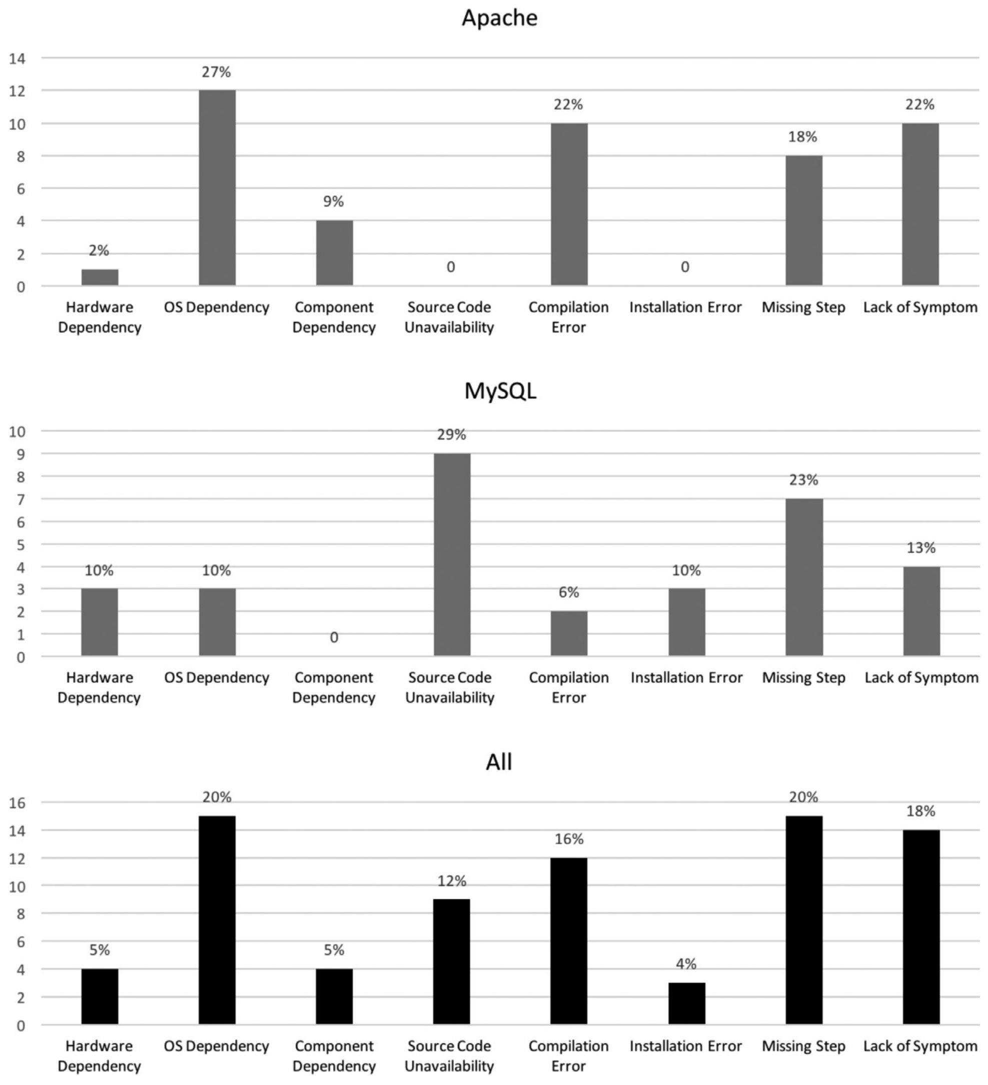

Fig. 3. Bug reproduction failure factor distribution.

Finding 10: Among all failed-to-reproduce bugs, the majority (74%) of them are due to OS dependency (20%), compilation error (16%), missing step (20%), and lack of symptom (18%).

Hardware dependency refers to performance bugs that can manifest themselves with only specific hardware resources. For example, reproducing MySQL bug #51325 requires 40 GB of memory to be configured for the configuration option `innodb_buffer_pool_size`. However, the required memory size exceeds the total amount of memory in our machine.

OS dependency refers to performance bugs that are operating system (OS) dependent, thereby failed-to-reproduce under our available OS (i.e., Ubuntu Linux). In several cases of our study, we have no access to the OSs described in the bug reports, such as Microsoft Windows and Mac OS X. For example, in Apache bug #56271, high memory consumption is observed on a Windows Server 2008 machine. We had no success in reproducing this bug on Linux; we suspect that exposing this performance bug requires calling OS-specific services (e.g., system calls).

Component dependency refers to performance bugs that are dependent on external software components but cannot be setup in our environment. For example, in Apache bug #38602, JBoss v3.2 is required to verify if Apache keeps sockets open when `KeepAlive` configuration option is set to *on*. Since JBoss is no longer free, we are not able to have it installed for reproducing the bug.

Source code unavailability refers to the version of a program cannot be retrieved from the source code distribution archives (apacheArchive, 2017; mysqlArchive, 2017). For instance, in MySQL bug #30414, the specific versions (e.g., v5.1.20, v5.1.21) are not found on the official distribution archive site.

Compilation error refers to the situation that the program fails to be compiled due to unsolvable compiler flags and/or library dependencies. For instance, in Apache bug #37680, make fails due to a missed library `libexpat.so.0`.

Installation error refers to the case that a program fails to be installed due to reasons such as the installation utility cannot locate the files to be deployed. For instance, in MySQL bug #15811, when executing `make install`, it reports an error message "recipe for target install-pkg include HEADERS failed". This is because the installation cannot locate certain header files.

Missing step refers to the lack of information on the steps of reproducing performance bugs. Ideally, we want to repeat the steps exactly as what are described in the bug report. Unfortunately, bug reporters tend to make optimistic assumptions about the expertise of bug report readers and often skip some critical steps for reproducing the performance bug. For instance, in Apache bug #43238, the bug reporter suggests to benchmark the Apache server with HTTPS requests. However, the specific approach to benchmark the web server is not described. Sometimes instructions on how to observe the symptom are unclear. For example, in Apache bug #48215, the extra negotiation of SSL connection is being reported, but it is not clear in the bug description on how to observe such behavior. Some would argue that a web debugging proxy utility such as Fiddler can be used, however, if the bug report could provide clear instructions, the extra research on setting up may have been avoided to cause further confusion.

Lack of symptom refers to when the expected symptom is not observed. For instance, in Apache bug #38737, the bug report describes a stall during server shutdown but we are not able to observe the stall in any of the processes. Unlike functional bugs, we cannot examine the expected program behavior by looking at the program output. To determine if a program has performance issues, we instead rely on performance bug symptoms such as long response time, a low throughput, or excessive use of system resources. Sometimes the level of magnitude is inconsistent with the symptom being reported. For example, in Apache bug #44026, when the web server is configured as a forward proxy, it should exhaust all available memory after a few thousand requests. However, in our experiment, we only observed a slight memory increase even after millions of requests are made. In any case, it is difficult for others to confirm the existence of a performance bug.

Fig. 3 summarizes the categories and the total number of bug reports falling into each category. As the results show, missing step, OS dependency, and lack of symptom are the top three factors leading to the failure of reproduction over the total number of failed-to-reproduce performance bug reports. The results also indicate that the factors vary across different subject programs. For example, source code unavailability is a major factor in MySQL but not in Apache. We conjecture that the reason is that code release policy differs across organizations. Table 4 describes 24 representatives failed-to-reproduce performance bug reports under each factor.

4.3. RQ3: Workaround the issues in failed-to-reproduce performance bug reports

Given the challenges of reproducing performance bug reports, we next describe the strategies we employed to increase the success of bug reproduction.

4.3.1. Hardware dependency

It is not always possible to have the exact same hardware settings as the original bug report. Our experience shows it is not always necessary either. In Apache bug #44026, it is reported that the reverse proxy server exhausted 16 GB of memory, but we only have 4 GB of memory on our machine. We are still able to reproduce this bug as long as we can observe the symptom that all 4 GB memory is exhausted. This implies that, in certain cases, we do not have to be restricted to the hardware settings stated in the bug report for bug reproduction.

4.3.2. OS dependency

If a performance bug does not require a specific version of OS, it is possible to use a different OS in the same family. For instance, Apache bug #37680 is reported on Fedora Linux. Although our OS is Ubuntu, we can still reproduce the bug because both OSs are based on Linux and the bug does not require a specific functionality provided by Fedora. Another example is Apache bug #45445, while the bug report states that Windows Server 2003 is needed to reproduce the bug, other Windows systems such as Windows XP can also be used for the bug reproduction as commented in the report. On the other hand, if the performance bug depends on features in a specific OS, the bug is unlikely to be reproduced. For example, reproducing Apache bug #18526 requires the process prioritization component that is only provided by OS X.

4.3.3. Component dependency

A bug report may not contain information about the dependent software components. For instance, Apache bug #27106 does not mention which version of OpenSSL is used. If we use the latest version, it may not have good backward compatibility. In addition, if a bug is triggered under a specific version of its dependent component, using a different version may not be able to expose the bug. Our solution is to find out the timeline of the bug report and retrieve the component version within the same time period. For example, in Apache bug #27106, exposing the performance bug requires installing OpenSSL, whose version is not mentioned in the bug report. Since the bug happens on Apache v2.0.48, which was released in October 2003, we can narrow down the range of the OpenSSL versions and use OpenSSL v0.9.7 to successfully reproduce the bug.

4.3.4. Source code unavailability

In a bug report description, the specific source code version might not be available. This problem can often be solved by using the source code of a previous version. Since a performance bug may not catch developers' attention immediately, the bug is unlikely to be fixed right away. This can make the bug appear in multiple versions prior to the reported program version. For instance, Apache bug #54852 is reported to exist in v2.2.x prior to v2.2.24, so we can select any version in v2.2.x to reproduce the bug. As another solution, if the faulty version is not available but its fixed version and code patch are available, we can restore the faulty version from the fixed version. In Apache bug #48024, the fix is introduced in its v2.4.x version, and by removing the patched code, we are able to generate a faulty version and reproduce the bug.

4.3.5. Compilation error

In large-scale software projects, the compilation is typically done through build utilities such as `configure` and `CMake` for C/C++ programs. A build error can sometimes be fixed by modifying the program source code. For instance, in Apache bug #27106, the compilation fails because of a compatibility issue on x86_64 machines for Apache v2.0.48. The solution is to change `APR_HAVE_SCTP=1` to `APR_HAVE_SCTP=0` in apr.h. Another solution to workaround compilation error is to install a pre-built binary distribution.

4.3.6. Installation error

Installation is the last step towards completing environment setup. Installation may fail due to the lack of permission to deploy files to a privileged directory. In such cases, on the Linux system, the root permission is normally required. In other situations such as the source distribution cannot locate header files as we have seen in MySQL bug #74325, we workaround this problem by deploying the database server to a SQL directory that contains the needed files.

4.3.7. Missing step

For example, in Apache bug #48024, the SED module consumes excessive memory when handling a file with long characters on a single line. To reproduce this bug, we need a back-end server that sits behind a reverse proxy, but details of what to be used as a back-end server are left out. To address this problem, we make an assumption that any web server that can serve HTTP requests could be used as a back-end web server. Therefore, we searched for

Table 4
Failed-to-reproduce performance bugs.

Problem	Subject	BugID	Bug Description & Explanation
Hardware Dependency	MySQL	61188	Slow performance on dropping compressed tables The bug requires 20 GB of memory to manifest
	MySQL	64258	High read timeout on InnoDB engine causes longer mutex wait Lack of SSD on dev environment
	Apache	24448	Java applet consumes significant CPU while Apache used as a proxy Bug requires a hardware device to host backend server
OS Dependency	MySQL	52102	InnoDB plugs has worse performance than built-in InnoDB engine Bug requires Microsoft Windows
	MySQL	18526	Thread priority is enabled by default on OS X which lowers performance Bug requires OS X
Component Dependency	Apache	38602	Web server does not keep HTTP connections alive JBoss v3.2 is not available
	Apache	45834	Authentication takes up to 15 mins to finish with mo_authnz_ldap module The firewall that sits between servers is unknown
Source Code Unavailability	MySQL	26079	Database hangs during binlog rotation when InnoDB engined is used MySQL v5.1.14 is not available
	MySQL	30414	Performance regression in throughput tests when logging is enabled MySQL v5.1.21 and v5.1.20 are not available
Compilation Error	Apache	35686	Memory leak due to the multi-threaded MPM worker module Apache fails to build with OpenSSL
	Apache	12757	LDAP cache fails to create cache file on all processes except the first one. GCC is not compatible with the source code
	Apache	38403	Child thread consumes 100% CPU as Apache used as a reverse proxy server Configure utility failed with an syntax error message
	MySQL	24148	Database hangs when closing SSL connections MySQL failed to recognize OpenSSL and crashed
Installation Error	MySQL	15811	Long execution time of insert statements with multi-byte character sets Installation errors out with recipe for target x failed
	MySQL	26527	SQL insertion with LOAD DATA INFILE is very slow in partitioned tables make install failed with message “recipe for target failed”
	MySQL	77094	System log buffer mutex contention Failed to install the specific sysbench version
Missing Step	Apache	45445	The connection timeout causes stalling on unreachable backend servers Bug requires a busy server with long-lived requests
	Apache	22106	Embedded SSI slows down web pages Lack of information on bug reproduction steps
	MySQL	27501	A significant increase in kernel time due to excessive getrusage() calls Steps to reproduce the bug are very limited
	MySQL	38551	Query cache consumes CPU time even when it is turned off Lack of instructions on how to trigger the bug
Lack of Symptom	Apache	44026	Server memory surges to 16 GB when used as forward proxy The expected level of memory usage is not observed
	MySQL	15815	Queries take significant longer if multiple queries are running concurrently Linear time instead of exponential decay is observed
	MySQL	20876	CPU spikes when creating 5k+ tables with large FIL_SYSTEM_HASH_SIZE Cannot observe the difference by adjusting option values
	MySQL	39253	Large query cache causes extended blocked mutex wait time Cannot observe the symptom specified in the bug report

“simple web server” and used the `SimpleHTTPServer` module from Python as the back-end web server. The performance bug was finally successfully reproduced. The take-home message is that, to reproduce bugs in server applications, depending on the type of components that are not provided, we might easily find substitutes to workaround the issue.

4.3.8. *Lack of symptom*

Performance bug symptoms describe the expected output we wish to observe when reproducing a performance bug. In many cases, however, we are not able to observe the symptoms. For instance, in Apache bug #38017, the web server is used as a reverse proxy but fails to serve content from cache, and thus causes a performance slowdown. The bug report suggests searching for a “_default_” string in the log, which is an indicator of this performance bug. However, we do not find this string in the log generated from our environment. As an alternative solution, we monitor the HTTP response status in the log and search for an HTTP status code 304, which is also an indicator that the cached content is not modified (apacheCaching, 2017). Not all performance bug symptoms are permanent. For example, in Apache bug #48024, a CPU spike only appears after requesting a large file and returns back to the normal level. The level of CPU usage is unnecessarily high and could lead to more serious problems on a busy server. It is difficult to notice the symptom without any external tools. To handle this problem, we leverage the Linux top command and record CPU utilization periodically to observe the bug symptom.

Table 5 provides a quick reference to the problems of performance bug report reproduction and their solutions. Table 6 reports the effectiveness of workarounds applied to the failed-to-reproduce bug reports for the eight failing factors. Column #Failed of Table 6 lists the number of (initially) failed-to-reproduce bug reports falling into each category. Column #Workaround lists the number of bug reports that workarounds have been applied. Column Suc. Rate reports the success rate of workarounds applied to the failed-to-reproduce reports. For example, among 18 failed-to-reproduce bug reports requiring specific OSs that we do not have in our environment, we fixed three of them and thus the success rate is 17%. Column #Reproduced reports the number of bugs that can be successfully reproduced with workarounds. Note that when a workaround has been applied to a bug report in one step does not imply the bug can be successfully reproduced be-

Table 5
Performance bug reproduction problems and suggestions

Problem	Suggestions
Hardware Dependency	Hardware limitation: adjust system resource to be used in proportion to the bug report specification. In MySQL bug #51325, the buffer pool is set to 20 GB and 40 GB respectfully. It is advised to allocate 80% of the system memory to the buffer. Accordingly, we use 1.5 GB and 3 GB on a machine that has 4 GB memory.
OS Dependency	OS not available: choose an alternative distribution in the same operating system family. In some cases, bugs reported on a specific Linux system can be run on a different Linux distribution. For instance, in Apache bug #38602, v2.2 can also run on Ubuntu although the bug is originally reported on RedHat.
Component Dependency	Missing the application version: sometime when the exact application version is not available in the bug report, we can use the timestamp on the bug report against the timeline of when each version is made available to reduce the scope of application versions that we must try.
Source Code Unavailability	Source code unavailable: restore the faulty version if a patch and a working version are available. In Apache bug #48024, the exact server version is not available to download. Instead, we know that a patch has been applied to v2.4, and by removing the patch from this version, we can reconstruct the faulty version.
Compilation Error	Error with online solution: adjust source code and makefile; Error without online solution: use a pre-built binary distribution. In MySQL bug #54989, when we try to compile executables from the source code, we received a CMake error message with no online solutions available. Since the offending source code is of not special interest in our investigation, we use a pre-compiled binary distribution instead.
Installation Error	Missing files during installation: try to skip deploying non-essential files. For instance, when we install openssl v0.9.7, the installation failed due to the manual file cannot be found. Since the manual is not essential to our purpose, we choose to install without manual file.
Missing Step	Vague description: follow through the report discussion. Missing workload instructions: synthesize a load simulation targeting specific requirements. To simulate a long running request, telnet is used in reproducing Apache bug #43081.
Lack of Symptom	Fail to observe symptoms: find alternative bug indicators. In Apache bug #38017, it is suggested that a "_default_" string should be searched in the log as an evidence for the miss cache hit performance bug. Since we can not find this string, instead we monitor HTTP status code 304 to confirm that content is served form the cache.

Table 6
Workaround efficiency and effectiveness.

Problem	#Failed	#Workaround	Suc. Rate	#Reproduced	Est. Effort
Hardware Dependency	5	1	20%	0	1 to 2 h
OS Dependency	18	3	17%	0	1 to 2 h
Component Dependency	8	1	13%	0	3 to 5 h
Unavailable Source Code	10	5	50%	5	1 to 2 h
Compilation Error	17	5	29%	5	1 to 5 h
Installation Error	4	1	25%	1	1 to 5 h
Missing Step	20	5	25%	5	3 to 5 h
Lack of Symptom	14	1	7%	1	3 to 5 h

#Failed = Number of failed-to-reproduce performance bug reports.
#Workaround = Number of bug reports required workarounds.
Suc. Rate = Percentage of bug reports with successful workarounds.
# Reproduced = Number of reproduced performance bug reports.

cause it may encounter other problems that cannot be resolved. The last column reports an estimated researchers' effort in finding the workarounds.

Finding 11: A non-trivial portion (22.9%) of failed-to-reproduce performance bugs can be reproduced by applying workarounds.

5. Discussion

We share our experience in reproducing performance bug reports in two open source server applications. Specifically, we study eight major factors that make performance bug report reproduction difficult and summarize possible solutions to increase the success of the reproduction. In this section, we summarize the implications learned from our study. The first part is geared towards practitioners, since they reflect the state-of-the-art practices. The second part provides a roadmap for researchers who plan to develop new tools and techniques for addressing performance issues, especially in server applications.

5.1. Implications to researchers

5.1.1. Fine-grained techniques on detecting missing information in bug reports are needed

Existing research on characterizing and predicting missing information in bug reports has been focusing on understanding the description of bug reports. Chaparro et al. (2017) use machine learning to automatically predict if a bug report contains complete information for understanding and reproduction. Although completeness of bug report description is important, it may not be sufficient to reproduce performance bugs. Our results suggest that reproducing performance bugs can be affected by a variety of fine-grained factors (Section 4.2), such as environment and dependencies. When building prediction models, it helps to output a detailed level of what is missing to provide suggestions in improving the quality of the bug report.

5.1.2. Testing tools should consider input actions and orders

As our results (Finding 3) have shown, while a majority of server performance bugs require no more than one data input to trigger, exposing them does require multiple actions (Finding 6). It is also worth noting that the order of actions have an influence to performance bug reproduction (Finding 7). However, most existing performance testing techniques (Nistor et al., 2013b; Pradel et al., 2014) consider only single inputs or workload. New testing techniques to generate an effective sequence of input actions for detecting performance bugs is desired. One way to obtain these actions is from user manuals and bug systems.

5.1.3. Testing tools should consider configuration options

The current state of research in testing for performance bugs considers two major aspects – test inputs and test oracles (Nistor et al., 2013b; Pradel et al., 2014). However, our results (Finding 4) suggest that exposing bugs require both specific data inputs and configuration options. Therefore, we need configuration-aware techniques to test for performance bugs. One challenge in configuration-aware testing is that the space of possible unique configuration combinations grows exponentially with the number of available configuration options. To address this problem, testers often evaluate a representative sample of all possible configurations (Qu et al., 2008; Yilmaz et al., 2004). One possibility is to leverage existing static analysis (Lillack et al., 2014; Rabkin and Katz, 2011) to identify performance-sensitive configuration options based on code patterns. Such options can be used to guide performance testing. Our results also suggest that performance testing can focus on one or two configuration options (Finding 4).

5.1.4. Performance test oracles should cover various symptoms

Our results (Finding 8) suggest that many performance bugs manifest through transient symptoms (e.g., high CPU utilization and low cache hits). In contrast to permanent symptoms, where the application simply hangs or slows down, transient symptoms are difficult to handle. While runtime profilers can be used to capture such information, one challenge is that the transient symptom may not always be observable during the entire execution. Therefore, cost-effective sampling-based profiling techniques are needed to catch performance bugs with transient symptoms.

5.2. Implications to practitioners

Although our study is primarily focused on reproducing performance bugs from the perspective of researchers, our findings may also benefit practitioners concerning the quality of bugs and the allocations of bug resolution efforts.

5.2.1. Writing good quality bug reports is important

As the last column of Table 1 shows, there is not much improvement in reproducing performance bug reports over the years. The results suggest that better practice in writing reproducible performance bug reports is needed. We return to the results in Section 4.2 (Finding 10). Factors including OS dependency, reproduction description, compilation, and symptoms are especially important for creating reproducible performance bug reports. For example, to successfully reproduce a performance bug report, it often requires a number of steps to setup the environment (Finding 2). Describing these steps in a clear way is beneficial for performance bug reproduction. Better even, this should motivate developers to design and adopt approaches to enforce bug reports to contain what is considered to be necessary to reproduce a bug. Recent advances (Chaparro et al., 2017) in applying natural language processing techniques on bug report analysis may make it possible to automate the procedure to check the completeness of a bug report. By using machine learning techniques, such as the clustering method, performance bugs may be automatically assigned to different categories as discussed in Section 4.2. A set of predefined rules can be associated with each category. Such rules will be checked, for instance, when the bug is considered to be “Lack of Symptom”, the system can then suggest potential symptoms for this bug based on similar bugs that do have symptom descriptions in the same category.

5.2.2. Using alternative solutions when possible

As our results have shown, a non-trivial portion of the initial failed-to-reproduce bug reports can be reproduced with additional effort (Finding 9). This implies that when it is not possible to follow the exact descriptions in the bug report, it is acceptable to reproduce the bug with alternative methods. Table 6 also suggests that source code unavailability is the easiest to fix, whereas lack of symptom is the most difficult barrier to overcome. Therefore, practitioners can allocate their efforts to find workarounds according to the causes of the failed-to-reproduce performance bug reports.

6. Related work

6.1. Studies of bug reproducibility

There is a great deal of research on studying the reproducibility of bug reports (Chaparro et al., 2017; Cotroneo et al., 2016; Erfani Joorabchi et al., 2014; Frattini et al., 2016; Gray, 1986; Grottke et al., 2010; Grottke and Trivedi, 2005; Sahoo et al., 2010). Erfani Joorabchi et al. (2014) mined software repositories to compare the characteristics of non-reproducible bug reports, such as the number of authors, number of comments, and the bug status transitions, to other bug reports. They defined six common categories of bug reports based on non-reproducibility causes. Sahoo et al. (2010) conducted an empirical study on the characteristics of bugs that influence the reproducibility in the server production environment. They randomly selected and inspected a number of fixed bug reports to study bug characteristics, such as the number of inputs used to trigger a bug and the types of symptoms as bugs manifest. Based on their findings, they proposed automated approaches for bug diagnosis. Our study and Sahoo's work share similarities in that we both studied server applications, a set of confirmed bugs, the number of inputs to trigger a bug, and the bug symptoms. Cotroneo et al. (2016) conducted a comprehensive study on the characteristics of bug manifestation process. In the study, they identified major triggers (i.e. workload, application's state, execution environment, and user behavior) under which conditions a bug got activated and manifested as a failure. We also studied the input triggers required to manifest the performance bugs.

On the other hand, our work is different from prior work in several aspects. First, we focused on reproducing performance bugs, whereas the prior work studied the reproducibility of general bugs. Performance bugs are non-functional bugs — they output the right functional output but normally take a much longer time to finish. About half of the reproduced performance bugs require certain levels of workloads to manifest. Prior work did not consider the characteristics that are specific to performance bugs. Second, we focused on the study from the perspective of researchers who tried to replicate a known reproducible performance bug with only the description of a bug report. Therefore, we selected confirmed performance bug reports that are known to be reproducible by developers, whereas prior work had different target audiences of their studies. Third, prior work studied the characteristics of the bugs from bug reports without trying to actually reproduce them in real environment. In contrast, we got first-hand experience from the perspective of researchers, and went through all the steps necessary to actually execute and reproduce performance bugs, and hence we were able to deliver a reusable set of benchmarks that contain performance bugs. This also explains why Sahoo et al. found that nearly 82% of bug symptoms can be reproduced — many bugs may not actually be reproduced on researchers' side for bug selection.

Chaparro et al. (2017) utilized natural language processing and machine learning techniques to automatically identify if bug reports miss important information that can affect understandability and reproducibility. Their work focused on analyzing bug reports and selecting linguistic patterns as machine learning features to automate detection of missing information in a bug report. Our study gave insights on fine-grained categories of information that is necessary to present in a bug report to increase its chance to be reproduced. As a result, our findings can be used by similar machine learning techniques to improve their prediction accuracy.

Gray (1986) classified bugs into Bohrbugs that were easily reproduced with certain inputs and Heisenbugs that were not deterministically reproducible. Bohrbugs are "faults that are easily detected and fixed and for which the failure occurrences are easily reproduced." Bugs from our study are unlikely to fall into this category because as our study indicates, they are very challenging to reproduce. On the other hand, Mandelbugs refer to the type of bugs that are complex and non-deterministic. Our studied bugs may fall into the category of Mandelbugs.

Grottke and Trivedi (2005) re-defined the widely but inconsistently used software faults terms that are aging-related bugs: a type of bug that leads to a higher probability of resulting in a failure or performance degradation. Specifically, in the paper, they clarified the relationship and definitions for Bohrbugs, Mandelbugs, and Heisenbugs. Later work by Grottke et al. (2010) conducted an empirical study in NASA space mission system software. They investigated four fault types: Bohrbugs, non-aging-related Mandelbugs, aging-related bugs, and unknown bugs in on-board software faults reported from 18 past space missions, and whether the fault type was independent of characteristics, such as failure effect and failure risk in the space mission system software. Some bugs used in our study may fall into the category of aging-related bugs, which was defined as "faults that can potentially cause software aging, which result in an increased failure rate and/or degraded performance". For instance, in Apache bug #27106, there is a memory leak with the HTTP request. We consider this bug to be an aging-related bug.

Frattini et al. (2016) discussed the process and influential factors in bug manifestation. Specifically, they surveyed the taxonomy of bug reproducibility, described the procedure for manually analyzing a bug report for its reproducibility, and applied machine learning techniques to predict bug classifications. They manually examined if the report was a real bug, and if not, the bug was marked as "NOT_BUG" or "UNKNOWN". Next, for bugs that had sufficient information, the following was examined: inputs and the application configurations required for exposing the bug.

Our manual bug selection approach was similar to theirs as we also utilized the bug repository system to filter out unwanted types of bugs (e.g. the NOT BUG class). We also examined the bugs carefully to identify the inputs and workloads that were required to expose the performance bugs.

There are several differences between Frattini's work and our study. First, Frattini's work focused on studying two categories of factors affecting reproducibility, including *workload-dependent* and *environment-dependent*, whereas we have defined a larger set of categories, such as *component dependency* and *lack of symptom*. Moreover, as discussed earlier in this section, one uniqueness of our study is that we tried to actually reproduce the bugs, so we were able identify more factors influencing reproducibility. We also suggested workarounds to improve the bug reproduction success rate.

Cavezza et al. (2014) studied the dependency of environmental factors on the reproducibility of software failures in MySQL, such as memory occupation, disk usage, and level of concurrency. Their experiment demonstrated that by increasing the usage level of such factors (e.g. disk usage) can increase the chance of reproducing a software failure. The major difference between their work and our study is that Cavezza's study investigated specific aspects of reproduction (e.g., determinism, environmental factors) for bugs in general, whereas we systematically studied a set of fine-grained factors (e.g., input parameters, configurations, reproducing steps) affecting the reproducibility of performance bugs. In addition, we provided alternative solutions to workaround failed-to-reproduce performance bugs. On the other hand, factors studied in their work may also be applied to performance bugs, for example, a higher disk usage may lead to a performance bug.

6.2. Performance bug empirical studies

There has been some work on the empirical study for performance bugs (Han and Yu, 2016; Jin et al., 2012; Nistor et al., 2013a; Zaman et al., 2012). Jin et al. (2012) studied 110 performance bugs from five software projects. They studied how performance bugs were introduced, exposed, and fixed. They looked at the root causes of performance bugs and the code patches. By observing the code patterns that fixed performance bugs, they summarized 25 efficiency rules. They then used these rules to detect performance bugs based on pattern matching. Nistor et al. (2013a) conducted a study of over 600 bugs to compare and contrast different characteristics of discovering, reporting, and fixing between performance bugs and non-performance bugs. Their study provided empirical evidence on the importance and challenges of performance bugs. They focused on the way that bugs were discovered and reported, where the authors claimed that a large percentage of performance bugs were discovered with code reasoning (33.9–57.3%) and a much smaller portion (5.5–10.4%) of performance bugs were identified with profilers. They reported the complexity involved in the bug fixing and concluded that performance bugs were likely to be more challenging to fix.

Zaman et al. (2012) studied 400 randomly selected performance and non-performance bug reports in Firefox and Chrome. They quantified the study findings in four dimensions: the impact on stakeholders, the context of the bug, bug fixes, and bug fix validations. As a result, their study found that performance bugs were more difficult to handle than non-performance bugs. Han and Yu (2016) studied the characteristics of 113 performance bugs in highly-configurable systems. They categorized the causes and fixes in performance bugs. A highlight of their study was to point out that configuration options were often neglected in the performance testing although some configuration options can often cause performance bugs. While previous research provided insights on identifying the root causes of performance bugs and guidance on addressing performance bugs in general, they did not conduct the study by actually reproducing bugs from performance bug reports.

6.3. Performance debugging and testing

Several techniques in testing, debugging, fixing, and avoiding performance bugs have been proposed in recent literature (Grechanik et al., 2012; Han et al., 2012; Jovic et al., 2011; Nistor et al., 2013b; Pradel et al., 2014). Han et al. (2012) proposed

StackMine, a debugging technique to discover high-performance impact call sequences from numerous and complicated call stack traces. Jovic et al. (2011) introduced Lag Hunting, a method that monitors deployed interactive system behavior and provides a list of performance issues. The authors argued that the use of profilers would not work for detecting perceptible performance slowness in interactive applications. Instead, they measured the latency to catch perceptible performance problems. Pradel et al. (2014) designed a regression testing technique to generate performance test cases for thread-safe Java concurrent classes. Grechanik et al. (2012) proposed a test generation framework, FOREPOST, to associate test inputs with their performance loads. Execution traces were clustered and used to train a classification algorithm to generate rules that describe the semantic patterns of good test inputs. Nistor et al. (2013b) proposed an automated performance testing oracle by identifying nested loops whose computation has repetitive memory-access patterns. While the above techniques are inspiring and effective, they considered only data inputs. Our study acknowledged prior work and suggested that a significant portion of performance bugs were related to configurations, input actions, and the order of input actions. These factors should be considered when designing software testing and diagnosis tools.

7. Conclusions

We conducted a performance bug reproduction experiment from the bug tracking systems of two open-source server applications. We studied 93 performance bug reports. Our empirical study showed that the rate to successfully reproduce a performance bug report was low (81%). We first studied the characteristics of the 17 performance bugs that were successfully reproduced. We then identified eight major factors that led to the reproduction failures in the remaining 76 bugs. We provided a list of suggestions on how to improve the chance of reproducing performance bugs. Out of the 17 successfully reproduced performance bugs, 15 of them utilized our workaround strategies. Our study provided guidance and insights for researchers and practitioners on improving the quality of performance bug reports and designing testing and diagnosis tools for handling performance bugs.

Acknowledgment

This research is supported in part by the National Science Foundation grant CCF-1652149.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.jss.2019.06.100 .

References

Apache http Server Source Code Distributions, 2017. https://archive.apache.org/dist/httpd/.
Apache Caching Guide. 2017. https://httpd.apache.org/docs/2.4/caching.html.
Aranda, J., Venolia, G., 2009. The secret life of bugs: going past the errors and omissions in software repositories. In: Proceedings of the International Conference on Software Engineering.
Apache Software Fundation Bugzilla. 2016. https://bz.apache.org/bugzilla/.
Attariyan, M., Chow, M., Flinn, J., 2012. X-ray: automating root-cause diagnosis of performance anomalies in production software. In: Proceedings of the USENIX Conference on Operating Systems Design and Implementation.
Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P., 2009. Fair and balanced?: Bias in bug-fix datasets. In: European Software Engineering Conference.
Bugzilla Keyword Descriptions. 2016. https://bugzilla.mozilla.org/describekeywords.cgi.
Burnim, J., Juvekar, S., Sen, K., 2009. Wise: Automated test generation for worst-case complexity. In: Proceedings of the International Conference on Software Engineering.
Cavezza, D.G., Pietrantuono, R., Alonso, J., Russo, S., Trivedi, K.S., 2014. Reproducibility of environment-dependent software failures: an experience report. IEEE 25th International Symposium on Software Reliability Engineering.
Chaparro, O., Lu, J., Zampetti, F., Moreno, L., Di Penta, M., Marcus, A., Bavota, G., Ng, V., 2017. Detecting missing information in bug descriptions. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.
Cotroneo, D., Pietrantuono, R., Russo, S., Trivedi, K., 2016. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. J. Syst. Softw. 113, 27–43.
Dean, D.J., Nguyen, H., Gu, X., Zhang, H., Rhee, J., Arora, N., Jiang, G., 2014. Perfscope: practical online server performance bug inference in production cloud computing infrastructures. In: Proceedings of the ACM Symposium on Cloud Computing.
Erfani Joorabchi, M., Mirzaaghaei, M., Mesbah, A., 2014. Works for me! Characterizing non-reproducible bug reports. In: Proceedings of the 11th Working Conference on Mining Software Repositories.
Frattini, F., Pietrantuono, R., Russo, S., 2016. Reproducibility of software bugs. In: Principles of Performance and Reliability Modeling and Evaluation. Springer, pp. 551–565.
Github. 2008. https://github.com.
Gray, J., 1986. Why do computers stop and what can be done about it? In: Symposium on Reliability in Distributed Software and Database Systems. Los Angeles, CA, USA, pp. 3–12.
Grechanik, M., Fu, C., Xie, Q., 2012. Automatically finding performance problems with feedback-directed learning software testing. In: Proceedings of the International Conference on Software Engineering.
Grottke, M., Nikora, A.P., Trivedi, K.S., 2010. An empirical investigation of fault types in space mission system software. In: 2010 IEEE/IFIP international Conference on Dependable Systems & Networks (DSN). IEEE, pp. 447–456.
Grottke, M., Trivedi, K.S., 2005. A classification of software faults. J. Reliabil. Eng. Assoc. Jpn. 27 (7), 425–438.
Han, S., Dang, Y., Ge, S., Zhang, D., Xie, T., 2012. Performance debugging in the large via mining millions of stack traces. In: Proceedings of the International Conference on Software Engineering.
Han, X., Yu, T., 2016. An empirical study on performance bugs for highly configurable software systems. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement.
Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S., 2012. Understanding and detecting real-world performance bugs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.
Jovic, M., Adamoli, A., Hauswirth, M., 2011. Catch me if you can: performance bug detection in the wild. In: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications.
Lillack, M., Kästner, C., Bodden, E., 2014. Tracking load-time configuration options. In: Proceedings of International Conference on Automated Software Engineering.
Molyneaux, I., 2009. The Art of Application Performance Testing: Help for Programmers and Quality Assurance. O'Reilly Media, Inc..
Mysql Community Server (archived versions), 2017. https://downloads.mysql.com/archives/community/.
Mysql Bugs Home. 2016. https://bugs.mysql.com/.
Nistor, A., Jiang, T., Tan, L., 2013. Discovering, reporting, and fixing performance bugs. In: Proceedings of the International Conference on Mining Software Repositories.
Nistor, A., Song, L., Marinov, D., Lu, S., 2013. Toddler: detecting performance problems via similar memory-access patterns. In: Proceedings of the International Conference on Software Engineering.
Olivo, O., Dillig, I., Lin, C., 2015. Static detection of asymptotic performance bugs in collection traversals. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.
Pradel, M., Huggler, M., Gross, T.R., 2014. Performance regression testing of concurrent classes. In: Proceedings of the International Symposium on Software Testing and Analysis.
Qu, X., Cohen, M.B., Rothermel, G., 2008. Configuration-aware regression testing: an empirical study of sampling and prioritization. In: Proceedings of the International Symposium on Software Testing and Analysis.
Rabkin, A., Katz, R., 2011. Static extraction of program configuration options. In: Proceedings of the 33rd International Conference on Software Engineering.
Sahoo, S.K., Criswell, J., Adve, V., 2010. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In: Software Engineering, 2010 ACM/IEEE 32nd International Conference on.
Techtarget. 2006. https://searchdatacenter.techtarget.com/definition/workload.
Oracle VM VirtualBox, 2016. Virtualbox.
Wert, A., Happe, J., Happe, L., 2013. Supporting swift reaction: automatically uncovering performance problems by systematic experiments. In: Proceedings of the International Conference on Software Engineering.
Yilmaz, C., Cohen, M.B., Porter, A., 2004. Covering arrays for efficient fault characterization in complex configuration spaces. TSE 29 (4).
Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L.N., Pasupathy, S., 2011. An empirical study on configuration errors in commercial and open source systems. In: Proceedings of the ACM Symposium on Operating Systems Principles.
Zaman, S., Adams, B., Hassan, A.E., 2012. A qualitative study on performance bugs. In: Proceedings of the International Conference on Mining Software Repositories.

Xue Han is in his fifth year of Ph.D. study in the Computer Science Department, University of Kentucky. He is passionate about doing research in Software Testing, Search Based Software Testing, Performance Modeling, Program Analysis, Machine Learning, Natural Language Processing, and Data Mining. He has published several research papers in premier Software Engineering conferences. Before his Ph.D. study, he was working as a.NET Engineer.

Daniel Carroll was a undergraduate student in the Computer Science Department of University of Kentucky when this work was completed.

Tingting Yu is an assistant professor of Computer Science at University of Kentucky. She received her Ph.D degree from University of Nebraska-Lincoln in 2014. Her research interests include software engineering, software testing, concurrent systems, and cyber-physical systems. Dr. Yu received the ACM SIGSOFT Distinguished Paper Award in 2016. She was a recipient of the NSF Faculty CAREER Award in 2017.