2019

CONFPROFITT: A CONFIGURATION-AWARE PERFORMANCE PROFILING, TESTING, AND TUNING FRAMEWORK

Xue Han

Right click to open a feedback form in a new tab to let us know how this document benefits you.
STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student’s advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student’s thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Xue Han, Student

Dr. Tingting Yu, Major Professor

Dr. Mirek Truszczynski, Director of Graduate Studies
CONFPROFIT: A CONFIGURATION-AWARE PERFORMANCE PROFILING, TESTING, AND TUNING FRAMEWORK

DISSERTATION

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the College of Engineering at the University of Kentucky

By
Xue Han
Lexington, Kentucky
Director: Dr. Tingting Yu, Professor of Computer Science
Lexington, Kentucky
2019

Copyright © Xue Han 2019
https://orcid.org/0000-0002-2527-2497
ABSTRACT OF DISSERTATION

CONFPROFITT: A CONFIGURATION-AWARE PERFORMANCE PROFILING, TESTING, AND TUNING FRAMEWORK

Modern computer software systems are complicated. Developers can change the behavior of the software system through software configurations. The large number of configuration option and their interactions make the task of software tuning, testing, and debugging very challenging. Performance is one of the key aspects of non-functional qualities, where performance bugs can cause significant performance degradation and lead to poor user experience. However, performance bugs are difficult to expose, primarily because detecting them requires specific inputs, as well as specific configurations. While researchers have developed techniques to analyze, quantify, detect, and fix performance bugs, many of these techniques are not effective in highly-configurable systems. To improve the non-functional qualities of configurable software systems, testing engineers need to be able to understand the performance influence of configuration options, adjust the performance of a system under different configurations, and detect configuration-related performance bugs.

This research will provide an automated framework that allows engineers to effectively analyze performance-influence configuration options, detect performance bugs in highly-configurable software systems, and adjust configuration options to achieve higher long-term performance gains. To understand real-world performance bugs in highly-configurable software systems, we first perform a performance bug characteristics study from three large-scale open-source projects. Many researchers have studied the characteristics of performance bugs from the bug report but few have reported what the experience is when trying to replicate confirmed performance bugs from the perspective of non-domain experts such as researchers. This study is meant to report the challenges and potential workaround to replicate confirmed performance bugs. We also want to share a performance benchmark to provide real-world performance bugs for evaluate future performance testing techniques. Inspired by our performance bug study, we propose a performance profiling approach that can help developers to understand how configuration options and their interactions can influence the performance of a system. The approach uses a combination of dynamic analysis and machine learning techniques, together with configuration sampling techniques, to profile the program execution, analyze configuration options relevant to performance. Next, the framework leverages natural language processing and information retrieval techniques to automatically generate test inputs and configurations to
expose performance bugs. Finally, the framework combines reinforcement learning and dynamic state reduction techniques to guide subject application towards achieving higher long-term performance gains.

KEYWORDS: Configurable Software System, Performance Testing, Performance Bugs, Performance Tuning
CONFPFIT: A CONFIGURATION-AWARE PERFORMANCE PROFILING,
TESTING, AND TUNING FRAMEWORK

By
Xue Han

Dr. Tingting Yu
Director of Dissertation

Dr. Mirek Truszczynski
Director of Graduate Studies

07/14/2019
Date
DEDICATION

To Tudun.
I would like to thank everyone who helps me out during my Ph.D. study. This dissertation would not have been possible without your encouragement, support, and kindness.

I would like to give special thanks to my advisor Dr. Tingting Yu for all her support in my research. Dr. Yu has been advising and mentoring me in every step of the way. She sets a great example of the type of research scientists that we all can look up to. I am deeply grateful for the countless hours she has spent with me on my research writing and presentations.

I want to thank my vice committee chair Dr. Jane Hayes for the memorable projects, seminars, and E-Day events we have done together. I want to thank Dr. James Griffioen and Dr. Arnold Stromberg for serving on my committee, and providing insightful comments and feedbacks during my Ph.D. comprehensive exam and defense. I want to thank Dr. Joseph Straley to serve as the external committee member, and to provide many interesting feedbacks and comments on my dissertation.

I also want to thank my DGS Dr. Mirek Truszczynski to offer me the teaching assistant position and to provide encouraging feedbacks on my Ph.D. study progress. I want to thank Dr. Yi Pike for her support while I worked as a TA. I would like to thank my collaborators, Dr. David Lo, Dr. Michael Pradel, Mr. Daniel Carroll for their patience and for having faith in me.

I want to thank all my colleagues and friends at the University of Kentucky to make it such an enjoyable journey for me.
And lastly, I want to thank my parents and my wife Yuanyuan for their unconditional support and understanding.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS ......................................................................................... iii

LIST OF TABLES ..................................................................................................... viii

LIST OF FIGURES .................................................................................................... ix

1 Introduction ........................................................................................................ 1
   1.1 A Summary of the CONFProfiTT Framework ............................................... 3
   1.2 Contributions .................................................................................................. 4

2 Background and Related Work ........................................................................... 5
   2.1 Background ...................................................................................................... 5
      2.1.1 Dynamic Analysis ..................................................................................... 5
      2.1.2 Performance Pattern ................................................................................ 6
      2.1.3 Test Specification Language ....................................................................... 6
      2.1.4 Natural Language Processing .................................................................... 7
      2.1.5 Reinforcement Learning .......................................................................... 7
   2.2 Related Work ................................................................................................... 8
      2.2.1 Configuration Testing ................................................................................ 8
      2.2.2 Performance Bug Study .......................................................................... 8
      2.2.3 Performance Modeling .......................................................................... 9
      2.2.4 Natural Language Processing .................................................................. 10
      2.2.5 Reinforcement Learning Techniques ...................................................... 10

3 Performance Bug Study ....................................................................................... 11
   3.1 Introduction ..................................................................................................... 11
   3.2 Motivation ....................................................................................................... 12
   3.3 Case Study ...................................................................................................... 15
      3.3.1 Data Sets .................................................................................................. 16
         3.3.1.1 Studied Subjects ................................................................................ 16
         3.3.1.2 Data Collection .................................................................................. 16
      3.3.2 Threats to Validity .................................................................................... 17
   3.4 Results ............................................................................................................. 18
      3.4.1 RQ1: Prevalence of Bugs ....................................................................... 18
      3.4.2 RQ2: Configuration Types ....................................................................... 19
         3.4.2.1 Parameter Configurations .................................................................. 19
         3.4.2.2 Hidden Configurations ...................................................................... 22
         3.4.2.3 System-Level Configurations ........................................................... 22
      3.4.3 RQ3: Causes of Bugs .............................................................................. 23
         3.4.3.1 When are Bugs Introduced? ................................................................. 23
3.4.3.2 Why do Performance Bugs Happen? ......................................................... 24
3.4.3.3 What configuration options are culprits? ..................................................... 25
3.4.4 RQ4: Bug Fixes .............................................................................................. 25
3.4.4.1 How to Fix Performance Bugs? ................................................................. 26
3.4.4.2 Are Patches Complex? .............................................................................. 26
3.5 Discussion ....................................................................................................... 26
3.5.1 Implications to Practitioners ....................................................................... 26
3.5.2 Implications to Researchers ....................................................................... 27
3.6 Conclusion ...................................................................................................... 29

4 Performance Bug Replication .......................................................................... 30
4.1 Introduction ..................................................................................................... 30
4.2 Motivating Examples ...................................................................................... 33
4.3 Case Study ....................................................................................................... 35
4.3.1 Data Sets ....................................................................................................... 35
4.3.1.1 Studied Subjects ...................................................................................... 35
4.3.1.2 Data Collection ...................................................................................... 35
4.3.1.3 Study Setup ............................................................................................ 36
4.3.2 Threats to Validity ....................................................................................... 38
4.4 Results ............................................................................................................ 39
4.4.1 RQ1: Reproduced Bug Reports and Their Characteristics ......................... 39
4.4.2 RQ2: Factors Leading to Failed to Reproduce Performance Bug Reports ..... 44
4.4.3 RQ3: Workaround the Issues in Failed-to-Reproduce Performance Bug Reports ........................................................................................................ 49
4.4.3.1 Hardware Dependency ............................................................................ 49
4.4.3.2 OS Dependency ...................................................................................... 49
4.4.3.3 Component Dependency ....................................................................... 49
4.4.3.4 Source Code Unavailability .................................................................. 50
4.4.3.5 Compilation Error .................................................................................. 50
4.4.3.6 Installation Error .................................................................................... 50
4.4.3.7 Missing Step ........................................................................................... 50
4.4.3.8 Lack of Symptom ................................................................................... 51
4.5 Discussion ....................................................................................................... 52
4.5.1 Implications to Researchers ....................................................................... 52
4.5.2 Implications to Practitioners ....................................................................... 53
4.6 Related Work ................................................................................................. 54
4.7 Conclusions ................................................................................................... 58

5 Performance Profiling ....................................................................................... 59
5.1 Introduction ..................................................................................................... 59
5.2 Background .................................................................................................... 62
6.6.2 Results and Analysis ........................................................................................................ 97

6.7 Discussion ............................................................................................................................... 100

6.8 Related Work ........................................................................................................................... 101

6.9 Conclusions ............................................................................................................................. 102

7 Performance Tuning ..................................................................................................................... 104

7.1 Introduction .............................................................................................................................. 104

7.2 Background ............................................................................................................................. 106

7.2.1 Tuning Performance Configuration .................................................................................. 106

7.2.2 Reinforcement Learning Techniques ................................................................................. 107

7.3 Problem Formulation ............................................................................................................... 108

7.4 Design of ConfRL .................................................................................................................. 111

7.4.1 The ConfRL Algorithm ...................................................................................................... 111

7.4.2 Ranking Performance-Influential Configurations ............................................................... 112

7.4.3 Generating Option Values .................................................................................................. 113

7.4.4 Reducing Runtime States ................................................................................................. 114

7.4.5 Measuring Performance ..................................................................................................... 114

7.4.6 A RL Running Example ..................................................................................................... 115

7.5 Empirical Study ...................................................................................................................... 116

7.5.1 Implementation .................................................................................................................. 116

7.5.2 Subject Programs ................................................................................................................ 116

7.5.3 Experiment Design ............................................................................................................. 117

7.5.3.1 Baseline Techniques ....................................................................................................... 117

7.5.3.2 RQ1: Effectiveness of ConfRL ....................................................................................... 117

7.5.3.3 RQ2: Efficiency of ConfRL ........................................................................................... 118

7.6 Results and Analysis ............................................................................................................... 119

7.6.1 RQ1 Effectiveness of ConfRL ............................................................................................ 119

7.6.2 RQ2 Efficiency of ConfRL ................................................................................................. 120

7.7 Discussion ............................................................................................................................... 121

7.7.1 Sensitivity of Strength ......................................................................................................... 121

7.7.2 Threats to Validity .............................................................................................................. 122

7.7.3 Limitations ........................................................................................................................... 122

7.7.4 Related Work ....................................................................................................................... 123

7.8 Conclusions .............................................................................................................................. 124

8 Conclusion and Future Work ...................................................................................................... 126

BIBLIOGRAPHY ............................................................................................................................ 128

VITA ................................................................................................................................................ 145
Table 3.1  Subjects and Characteristics .................................................................16
Table 3.2  Types of Configurations ......................................................................19
Table 3.3  Parameter Configurations ..................................................................21
Table 3.4  Types of Configurations ......................................................................21
Table 3.5  System-Specific Configurations ............................................................21
Table 3.6  Stages of Bugs ...................................................................................24
Table 3.7  Causes of Bugs....................................................................................24
Table 3.8  Bug Fixes .............................................................................................25
Table 4.1  Subject Characteristics ......................................................................39
Table 4.2  Reproduced Bugs and Their Characteristics ........................................40
Table 4.3  Workload Type ..................................................................................42
Table 4.4  Performance Bugs Failed-To-Reproduce ..............................................45
Table 4.5  Performance Bug Reproduction Problems and Suggestions ....................48
Table 4.6  Workaround Efficiency and Effectiveness ............................................52
Table 5.1  Subjects and Their Characteristics .......................................................71
Table 5.2  Performance-Influencing Configuration Options Ranking .....................74
Table 5.3  Location-Level Complexity Models .....................................................75
Table 6.1  Workaround Efficiency and Effectiveness ............................................85
Table 6.2  Number of Patterns to Detect Frame Elements .....................................88
Table 6.3  RQ1: Test Frame Extraction Accuracy .................................................97
Table 6.4  RQ2: Performance Test Frame Generation .........................................98
Table 6.5  RQ3: Performance Testing with GA ....................................................99
Table 7.1  Configuration Option .........................................................................109
Table 7.2  Apache RL States ...............................................................................110
Table 7.3  State-Action Table ...........................................................................110
Table 7.4  Characteristics of Subjects ..................................................................117
Table 7.5  Effectiveness and Efficiency of ConfRL .............................................121
Table 7.6  Impact of State Reduction Techniques ...............................................121
LIST OF FIGURES

Figure 3.1  Apache Bug #54852 ................................................................. 13
Figure 3.2  MySQL Bug #72703 ................................................................. 13
Figure 3.3  MySQL Bug #77094 ................................................................. 14
Figure 3.4  Apache Bug #46749 ................................................................. 14
Figure 3.5  Apache Bug #42829 ................................................................. 15
Figure 3.6  Firefox Bug #415489 ............................................................... 15
Figure 3.7  Bug Classification ................................................................. 19
Figure 4.1  Reproducing Apache Bug #48024 ........................................ 41
Figure 4.2  Order of Input Actions ........................................................... 43
Figure 4.3  Bug Reproduction Failure Factor Distribution ....................... 45
Figure 5.1  Apache Bug #54852 ................................................................. 61
Figure 5.2  Overview of CoProf ............................................................... 64
Figure 5.3  Code Location Errors on AE-S1 and PS-S1 ......................... 77
Figure 6.1  MySQL Bug #74325 ............................................................... 84
Figure 6.2  Overview of PerfLearner ......................................................... 87
Figure 6.3  A Common Pattern to Identify Command ............................. 89
Figure 6.4  A Common Pattern to Identify Data Inputs ............................ 89
Figure 6.5  A Common Pattern to Determine A Workload ...................... 91
Figure 6.6  An Example of Performance Test Frame Generation .......... 92
Figure 6.7  API Description for innodb_fill_factor ............................... 92
Figure 6.8  Test Frame Generation ............................................................ 98
Figure 6.9  Weight Sensitivity Analysis .................................................... 100
Figure 7.1  Weight Sensitivity Analysis .................................................... 107
Figure 7.2  ConfRL Overview ................................................................. 109
Figure 7.3  Reinforcement Learning Plots ................................................. 120
Modern software systems are highly-configurable, allowing users to customize a large number of configuration options while retaining a core set of functionality. We consider a system highly-configurable if it comes with at least dozens of configuration options that control its core and add-on functionalities. The environment to which the application deploys has become increasingly more complex. The application can frequently interact with other system components such as shared libraries, environment variables, and kernel modules. The complexity of the configuration space and the sophisticated constraints among configuration settings could easily cause performance bugs. We refer to a performance bug as inefficient code sequences that can cause significant performance degradation and resource waste [72]. Unlike a functional bug that typically leads to system crashes or incorrect results, a performance bug can cause significant performance degradation, leading to problems such as poor user experience, long response time, and low system throughput [26, 72, 92].

Compared to functional bugs, performance bugs are substantially more difficult to handle because they often manifest themselves only with large inputs and specific execution environments [92, 96]. As such, traditional testing such as coverage-based approaches may not be effective. To address these problems, numerous research efforts have been made to analyze, detect, and fix performance bugs [27, 62, 72, 73, 93, 96]. For example, rule-based pattern matching has been used to detect performance anomalies under normal executions [72]. Several test case generation techniques have been proposed to generate large workload test inputs [27, 107].

However, most existing work assumes default configurations while ignoring the influence of various configurations. The tasks of performance testing and debugging can still be challenging without considering the complex combinations of configuration options. A typical configurable system may have thousands of options, and this generates numerous possible configurations. For example, Apache has more than 1000 possible configuration options [32]. Therefore, we believe that to answer research questions related to how configurations influence system performance can guide the design of, and improve techniques for addressing performance issues, including performance profiling, performance testing, and performance tuning.

Where performance profiling is concerned, although there have been techniques on building performance models for configurable software systems [59, 126, 127], they are black-box techniques and can have several limitations. First, they use randomly chosen configuration values and can expose performance-influence configuration options only when they are assigned to specific values (e.g., large workloads). Second, the use of random configuration space sampling often requires trying a large number of options to learn a performance model and thus can significantly slow down the learning process. Third, black-box techniques use configuration values and the performance measures to learn models, but the internal execution
profiles of the program are not visible. As such, developers may not be able to pinpoint the code locations that could lead to performance issues.

Profiling methods depend on the chosen set of input values, which is a known weakness [125] for successfully detecting performance bugs in the subject under test. To address this problem, several test case generation techniques have been proposed to generate large workload test inputs for increasing the chance of exposing performance bugs [27, 107]. However, there are several limitations in existing performance test generation techniques – many techniques focus on evolving the values of certain input parameters while keeping the other parameters as default. For example, Burnim et al. [27] focus on increasing the workload values of data inputs while keeping the values of configuration options as default. These techniques may be ineffective at detecting performance bugs due to combinatorial effects of different input parameters. For example, in Apache bug #52914, the performance bug is exposed only when the configuration options KeepAlive and RequestReadTimeout are specified. Otherwise, by using the default configuration, this performance bug cannot be triggered even if the workload (e.g., the number of requests) is increased.

While a full performance testing with all combinations of input parameters can address the above problem, it is infeasible due to the enormous combination space. For example, the latest version of Apache HTTP Server has 618 input parameters (610 configuration options and 8 types of data inputs). It is impractical to try all combinations of values for these input parameters. To reduce the cost of performance testing, Shen et al. [125] use a genetic algorithm (GA) as a search heuristic for obtaining combinations of input parameter values that maximize the execution time. However, this technique evolves all input parameters, which can be inefficient because many parameters may not provide contributions to the application’s performance.

Where performance testing is concerned, techniques must be able to increase the chance of exposing performance bugs. Given a large configuration space, software testing can be expensive. While sampling-based techniques have been proposed to reduce the cost of configuration testing [111, 159], they are not powerful enough to expose performance bugs. These techniques focus on achieving high coverage but exposing performance bugs usually requires specific input and configuration option values. In addition, performance test oracles are non-trivial because an increasing execution time is not a sufficient criterion. A performance bug can be exposed via a variety of symptoms, such as low CPU usage and long synchronization time.

Ideally, developers catch all bugs in the testing phase. When a performance problem occurs in production (e.g. a significant slowdown with HTTP responses in a web server), system administrators or developers need to reconfigure the system to find a configuration setting for
better performance. However, it is often not an easy task to figure out the best settings for a system with a large number of configuration options.

Even for domain experts, it is often not an easy task to configure the software system to get the best performance. For example, as one experienced user complained in Apache HBase Bug #13919, “There are current many settings that influence how/when an HBase client times out. This is hard to configure, hard to understand, and badly documented.” In addition, manually changing the configuration can be tedious, inefficient, and impractical. For instance, in the case of a web server, the volume of request level changes at different times of the day. It is not practical to ask administrators to change configuration settings to keep up with the level of web request changes. One way to tune configuration options for better performance is through reinforcement learning. The dynamic environment can be modeled as input workload from production environment (e.g., the number of HTTP requests), the states can be modeled as the current system configuration (i.e., a combination of configuration option values), the optimal control policy refers to a set of configuration options changes for achieving a better performance compared to the current state, and the reward value can be obtained by comparing the differences in performance measures before and after the configuration tuning. The benefit of RL is that it does not require domain knowledge of the system and is able to update optimal policies continuously in the long run.

1.1 A Summary of the CONFProfiTT Framework

CONFProfiTT provides a set of techniques to address the issues of performance profiling, performance testing, and performance tuning.

For performance profiling, the framework iteratively selects configuration options and utilizes combinatorial interaction testing, a configuration testing method to sample configuration option value. And finally, the framework builds a light-weight performance prediction model to promptly evaluate the performance impact on a given set of configuration option values.

For performance testing, we want to use data mining and natural language processing techniques to extract and analyze test frames from a performance bug. Since history repeats itself, we believe by extracting the test frame elements that are frequently involved in exposing performance bugs from past performance bug report is likely to be used in performance test cases to expose future performance bugs.

For performance tuning, the key idea is to use reinforcement learning (RL) techniques to automate performance configuration. RL is a process of learning by interactions with a dynamic environment, aiming to generate the optimal control policy for a given set of states guided by
the reward values. Therefore, we can formulate the task of tuning performance configuration as an RL problem, in which the optimal policy refers to a configuration generated for achieving higher performance with respect to the current system state. The reward value can be obtained by comparing the differences in performance measures before and after the configuration tuning.

1.2 Contributions

This dissertation makes the following contributions:

- We study the characteristics of the real world performance bugs to gain insights on what to use to portrait performance issues induced by configuration settings. We also study various known anti-performance patterns to advance our understanding of the root causes of performance bugs.
- We conduct an empirical study to report the first-hand experience on replicating performance bugs from the perspective of researchers.
- We design CoProf, a profiler that detects and ranks performance-influential configuration options.
- We propose a PerfLearner, a technique uses data mining and natural language processing to extract test frames from bug report to detect performance bugs.
- We develop an approach, ConfRL, that can automatically select and tune configuration options in response to the environment dynamics to achieve higher performance.
In this chapter we discuss the technical background used in the framework and we then discuss the related work accordingly.

2.1 Background

2.1.1 Dynamic Analysis

Dynamic program analysis requires the execution of application binaries. In general, dynamic analysis involves two phases: code instrumentation and trace analysis. The way instrumentation works is by inserting extra code on specific instructions to monitor program states during the runtime. Besides, some dynamic analysis tools offer functionalities on modifying instructions to alternate program runtime behavior as necessary. Although dynamic analysis incurs an execution overhead, it delivers much less false positives compared to the static analysis techniques. Some of the most commonly used dynamic analysis tools include the Intel Pin for C and C++ assembly code, ASM [15, 16, 23, 77], Soot [132], and BTrace [24] for Java bytecode.

Pin. One common way to help developers to locate performance culprits is to use profiling tools. However, profilers are known for its problems with false positives (procedures ranked high might be just some legit computationally expensive operations) and false negatives (with the “right” input, the low ranked procedures could potentially hurt performance) [93]. Besides, such utilities are worked in a black box manner. To get the maximum flexibility, custom profilers are often built with the help of dynamic instrumentation tools.

Luk et al. [84] presented a detailed discussion of using Pin as the instrumentation tool. Pin provides a rich API that developers can call from their C and C++ programs. The API offers a good level of abstraction to help developers focus on their tasks rather than distract them with concerns of handling different types of instruction sets on various architectures. A few APIs are, however, platform dependent. Pin also provides various instrumentation granularity. For instance, users can choose to instrument on the image level, procedure level, or the instruction level. Pin defers code discovery and automates optimization during the runtime instrumentation stage. These features distinguish itself among similar tools like Valgrind [143] and DynamoRIO [44].
2.1.2 Performance Pattern

**Program Performance Anti-Patterns.** Smith et al. [128, 129, 130] illustrated a series of anti-patterns that could lead to performance degradation. By examining the nature of each anti-pattern, the authors first described its impact on system performance and scenarios where such patterns occur. They then moved on to discuss strategies to minimize or even avoid the negative effect of such patterns. Unlike former studies targeted on reported bugs, this study provides insights from the perspective of software architecture design and best programming practices to circumvent performance loss. Those studied anti-patterns provide potential guidelines for rule-based [79] tools to discover numerous performance issues.

**ORM Anti-Patterns.** Multi-layered architecture design used in enterprise application has gained its popularity in the past decades. On the data access layer, it is convenient to adopt the abstraction of treating database tables as programming language objects. When mapping relational database objects into application code, code performance is not a primary concern. Chen et al. [31] proposed a framework to help developers find performance anti-patterns in Object-Relational Mapping (ORM) and suggest priority based on the severity of the found performance anti-pattern. In this paper, both the excessive data (doing unnecessary data retrieval for instance) and one-by-one (the operation overhead can be reduce by putting such queries in a batch) anti-patterns are examined. Utilizing taint analysis, the framework identified code path for data access and detected anti-pattern using data flow and rule-base approaches. Performance assessment was done through statistically evaluating the performance gain from before and after the anti-patterns fix. Naturally, an anti-pattern has a greater performance improvement will be given higher priority.

2.1.3 Test Specification Language

Test specification language (TSL) is created to define combinations of program input parameters and environmental factors. The concept of the test frame was first introduced in the category-partition method with TSL [97]. Each combination is a test frame that can be converted into actual test cases. A performance test frame consists of three input categories: command, configuration, and data input. A test frame can have one command in the command category, zero or more configuration options in the configuration category, and zero or more data inputs in the data input category. Each command, configuration option, and data input in a test frame is generally referred to as a test frame element or frame element.
2.1.4 Natural Language Processing

Natural language processing (NLP) is a subfield of computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human (natural) languages, in particular how to program computers to process and analyze large amounts of natural language data [152]. NLP algorithms split natural language text into tokens. When a NLP matching pattern is defined, programs can perform match not only on the textual tokens but also the part-of-speech as well. One popular use of NLP is to associate machine learning techniques while treating tokens as features. Natural language processing provides the foundation for information retrieval. Information retrieval (IR) is the activity of obtaining information system resources relevant to an information need from a collection. Searches can be based on full-text or other content-based indexing. Information retrieval is the science of searching for information in a document, searching for documents themselves, and also searching for metadata that describes data, and for databases of texts, images or sounds [151].

2.1.5 Reinforcement Learning

Reinforcement Learning (RL) is the procedure of learning from interactions between an agent and the environment to determine what is the best action to take under any given state to achieve the maximum long-term rewards [136].

**Markov Decision Process.** The basic form of a reinforcement learning problem is encapsulated as the Markov Decision Process (MDP). Formally, an MDP is used to describe an environment for reinforcement learning, where the environment is fully observable. The Markov Decision Process is consisted of a finite set of states, a finite set of actions, a state transition matrix: $P_{ss'} = P[S_{t+1} = s' | S_t = s]$ [115], a reward function $R$, and a discount factor $\gamma$. A state has Markov property if and only if each state captures the information from all past states that lead to the current state. A policy $\pi$ gives the probability to take an action given a state $s$: $\pi(\alpha | s) = P[A_t = \alpha | S_t = s]$. The action-value function $q_\pi(s, \alpha)$ is the expected total rewards given state $s$ by taking an action $\alpha$ following the policy $\pi$. The goal of solving the MDP problem is to find the optimal action-value function: $q_*(s, \alpha) = \max_\pi q_\pi(s, \alpha)$. In the Bellman Optimality Equation form, the action-value function can be written recursively as: $q_*(s, \alpha) = R_s^a + r \sum_{s' \in S} \max_\alpha q_*(s', \alpha')$.

**Model-Free RL.** If a problem can be modeled as the MDP, it can be solved analytically through value-iteration and policy-iteration algorithms. However, for larger problems, the inversion of the state transition matrix can be very expensive. Most real-world problems cannot be formulated as MDP since the environment is not fully observable. It is also difficult to
describe the rules in a dynamic environment, hence the MDP transition function is unknown. There are a set of techniques to estimate the action-value function of an unknown MDP, such methods are referred to as Model-Free reinforcement learning algorithms. Temporal-Difference (TD) learning is one such method. The Q-Learning method is one type of TD learning that is based off the Bellman Optimality Equation: \( q_s(s, a) = R_s + r \sum_{s' \in S} \max_{a'} q_s(s', a') \).

2.2 Related Work

2.2.1 Configuration Testing

There has been a great deal of work on configuration-aware techniques [17, 112, 160, 165]. For example, Yin et al. [160] study a number of configuration bugs to understand the configuration errors in commercial and open source systems. Rabkin et al. [112] propose a static analysis technique to extract configuration options from Java code. There has been a large body of work in the testing community that demonstrates the need for configuration-aware testing techniques and proposes methods to sample and prioritize the configuration space [111, 159]. Zhang et al. [165] have proposed a technique to diagnose crashing and non-crashing errors related to software misconfigurations. There has also been recent work that uses configurability as a way to avoid failures through self-adaptation [137]. However, none of these work deals with performance bugs.

2.2.2 Performance Bug Study

There has been some work on empirical study for performance bugs [72, 92, 163]. For example, Zaman et al. [163] study the bug reports for performance and nonperformance bugs in Firefox and Chrome. Their study found that performance bugs are more difficult to handle than non-performance bugs. Jin et al. [72] study 109 performance bugs from five software projects. While the above work provides insights and guidance on addressing performance bugs in general, it does not study in depth about performance bugs in highly-configurable software systems.

There has been recent work on testing, debugging, fixing and avoiding performance bugs [27, 62, 73, 93, 96]. For example, Nistor et al. [93] identify loops whose computation has repetitive memory-access patterns. StackMine mines call stack traces to discover call sequences with a high performance impact [62]. Pradel et al. [107] generate performance test cases. Grechanik et al. [54] select test cases for performance testing. While the above techniques are inspiring and effective, they assume the default configurations and do not consider performance bugs caused by configurations. Foo et al. [49] use ensemble learning techniques to detect
performance regressions due to environment-specific variations. Their work focus on system-specific configurations, whereas we have studied in depth a wide range of configurations.

2.2.3 Performance Modeling

From the performance modeling perspective, there has been much work on constructing performance models for various purposes [59, 78, 117, 126, 139], such as using learning approach to find performance influential configurations [126], creating performance models by profiling [78], and performance modeling by static and dynamic program analysis techniques [139]. All these techniques provide good insights about factors involved in the performance models. However, our study analyzes more thoroughly on how performance bugs are related to configurations, and thus complementary.

Various methods have been used in the computer science field for modeling system performance [35, 74, 106, 134]. Xiao et al. [155] present DeltaInference, a strategy for inferring workload-dependent performance bottlenecks. The core idea of the proposed method corresponds to two inference models, namely, the temporal inference and spatial inference. The temporal model is built on profile data from different workloads as input on a given program location, where the complexity model fits either a linear or power law regression function. The calling stack is used as the context information to aggregate same function calls under different program states in order to fine tune the quality of profile data for building the temporal model. The initial model is constructed based on a representative value range (RVR) and refined heuristically by extending the RVR to include data points outside the RVR zone to improve the robustness of the complexity model. In the spatial inference, the framework identifies complexity transitions to signal the finding of a potential performance bottleneck. For instance, when transiting from one program location to another, if the order of the complexity model raises, for example, from linear to an higher order, then there exists a complexity transaction.

Since the DeltaInference framework is primarily designed for workload sensitive graphical user interface applications, it is unclear if this method can be generalized on finding other types of performance bottlenecks. Plus, the discussion on how program locations are selected initially is rather vague. Obviously, if we treat every method as a candidate and build prediction models for temporal inference, the framework might not scale well with projects containing large numbers of methods.

Siegmund et al. [126] propose a performance influence model that associates performance with system configurations. This model helps users to understand the performance influence of individual configurations and interactions among configurations. The authors apply a linear regression model to predict the performance influence of system configurations. New configuration items are added to the model only if it can reduce the prediction error with
forward feature selection. The selected configurations are validated with a backward learning – it monitors whether the prediction error worsens by removing one configuration at a time.

2.2.4 Natural Language Processing

There has been considerable work on using natural language and information retrieval techniques to improve code documentation and understanding [30, 46, 64, 65] and to create code traceability links [7, 42, 99]. Ng et al. [30] propose a method using natural language processing and machine learning techniques to detect missing information from bug reports. Vijay-Shanker et al. [46] present an algorithm to split program identifiers into tokens by using the word frequencies mined from the program source code. Vijay-Shanker et al. [64] propose a tool utilizing natural language information to automatically select the appropriate expansion appears in the program identifier names. Vijay-Shanker et al. [65] present a method to mine words that are semantically similar in the software context to bridge the mismatch limited by synonyms.

2.2.5 Reinforcement Learning Techniques

Some literature [25, 113] explore the use of reinforcement learning in the context of dynamically adjusting resource allocations (e.g. CPU and Memory) on the resource sharing virtual machine environment. Such efforts are mainly focused on optimizing the hardware-level resource configurations on the virtual machine environment where guest systems may compete for shared resources. Bu et al. [25] propose RAC, a reinforcement learning approach to automatically update the application configuration in response to the web traffic and virtual machine changes. Rao et al. [113] propose a reinforcement learning approach to automatically configure resources on virtual machine (VM). In their work, the configuration space is defined in terms of the system resource allocations in the VM environment. The number of configuration options (CPU, MEM) to change is small.
Modern computer systems are highly-configurable, complicating the testing and debugging process. The sheer size of the configuration space makes the quality of software even harder to achieve. Performance is one of the key aspects of non-functional qualities, where performance bugs can cause significant performance degradation and lead to poor user experience. However, performance bugs are difficult to expose, primarily because detecting them requires specific inputs, as well as a specific execution environment (e.g., configurations). While researchers have developed techniques to analyze, quantify, detect, and fix performance bugs, we conjecture that many of these techniques may not be effective in highly-configurable systems. In this chapter, we study the challenges that configurability creates for handling performance bugs. We study 113 real-world performance bugs, randomly sampled from three highly-configurable open-source projects: Apache, MySQL, and Firefox. The findings of this study provide a set of lessons learned and guidance to aid practitioners and researchers to better handle performance bugs in highly-configurable software systems.

3.1 Introduction

A natural question to ask is to what extent do performance bugs have the potential to go undetected if tested under default environments. If only a few performance bugs involve configurations, then developers may use existing performance testing and modeling techniques and do not need to be overly concerned about dealing with configurations. On the other hand, if performance bugs are indeed sensitive to configurations, it is also worth exploring the characteristics of such configurations. For example, one may ask if certain configuration options (e.g., cache settings) are more likely to trigger performance bugs than others. If configuration-related performance bugs differ from those general performance bugs, developers may not use existing configuration-aware techniques.

In this chapter, we perform a characteristic study on performance bugs related to configurations. We consider performance bugs caused both by misconfigurations and those that can manifest under specific local and environmental configurations. We aim to uncover and quantify the extent to which the performance problem exists on real-world highly-configurable software systems. Specifically, we manually inspect 193 performance bugs from three popular and highly-configurable open-source applications: Apache, MySQL, and Firefox. We then perform a deep analysis on 113 configurations-related performance bugs. We see this study as a way to share with researchers and practitioners the performance issues that the configurability brings, a set of lessons learned, and a roadmap for developing configuration-aware techniques to address performance issues. In this chapter we have made the following contributions:
• We examine the prevalence of configuration issues that have led to performance bugs. We find that more than half of the performance bugs (59%) are due to configuration problems.
• We classify configurations into three types: parameter, hidden, and system-specific configurations. We find that a majority of configuration-related performance bugs (78% – 92%) are related to configuration parameter settings. However, a non-trivial portion of studied bugs (8% – 17%) are due to system-level configurations, motivating the need to consider system properties when addressing performance problems.
• We study different causes of configuration-related performance bugs. We find that performance bugs can be detected by observing internal symptoms, such as high CPU usage, low cache hit, and long synchronization time. These symptoms are also related to different domains of configuration options (e.g., memory, concurrency, and network). We also find that configuration-related performance bugs are caused by a small subset of the entire configuration space, motivating the use of configuration space reduction techniques to handle performance bugs.
• We examine the fixes of configuration-related performance bugs. We find that a majority of studied bugs (88%) require fixing source code instead of just changing values in configuration options. In addition, the patches for the examined bugs involve over 30 lines of code on average, which are more complex than the patches for general performance bugs (less than 10 lines) [72].
• Finally, we provide a set of lessons learned that will help practitioners and researchers better understand and address performance bugs in highly-configurable software systems.

3.2 Motivation

We define a configurable system as a software system with a core set of functionality and a set of add-on features, which are defined by a set of configuration options – including user provided options (or parameters) and environment settings (e.g., libraries, components, etc.). Changes to the value of configuration options affect programs’ behavior in some ways. We use the term “configuration options” and “configuration parameters” interchangeably.

We use six motivating examples from the bugs we have studied to answer the following questions:

1) Do performance bugs require specific configurations to manifest?
2) Are configuration-related performance bugs different from configuration-related general bugs for testing, debugging, and fixing?
3) Do developers need special tools to handle performance bugs?
Wasted Loop Computation. Some users occasionally experience a slowdown during the Apache graceful restart as stated in Apache bug #54852. To trigger this bug, user needs to start Apache with a relatively larger number for the StartServers option (e.g., StartServers = 60). In this case, the server takes more time to complete the restart than usual. A six-month code inspection reveals that the bug is caused by a wasted loop computation. When the children servers already exited, there is no need to iterate these servers. Figure 3.1 shows the fix for this bug. By checking the existence of the children server processes first, the time-consuming dummy_connection function is skipped if the server no longer exists. Although this bug is not due to misconfiguration, exposing the bug requires setting the configuration option (i.e., StartServers) to a specific value.

```c
ap_mpm_pod_killpg(ap_pod_t *pod, int num){
    for (i=0; i<num && rv==APR_SUCCESS;i++) {
        if(ap_scoreboard_image->servers[i][0].status!=SERVER_READY
            && ap_scoreboard_image->servers[i][0].pId
            == 0)
            continue;
        rv=dummy_connection(pod);
    }
}
```

Figure 3.1 Apache Bug #54852

Page Not Cleaned. The page cleaner feature (i.e., the innodb_page_cleaners option) can improve MySQL performance scalability by spawning multiple threads to flush dirty pages from buffer pool instances. In MySQL bug #72703, the database server experienced a slowdown during the shutdown phase. The root cause of this bug is due to the wrong implementation of the page cleaner feature. Figure 3.2 shows the fix associated with this option. This example indicates that incorrect implementation of configurations can negatively impact application performance.

```c
/*storage/innobase/srv/srv0start.cc*/
for (i = 1; i < srv_n_page_cleaners; ++i){
    os_thread_create(buf_flush_page,
        cleaner_worker, NULL, NULL);
}
```

Figure 3.2 MySQL Bug #72703

Lock Contention. In MySQL bug #77094, when innodb_flush_log_at_trx_commit is set to 2, the two logging functions (commit and write) both use the log_sys->mutex lock to write to a buffer. This inevitably causes lock contention that hurts the application performance. The fix is to use two different buffers, as shown in Figure 3.3, so that commit and write functions can be concurrently executed.
**Cache Not Purged.** In Apache bug #46749, a developer reports that when the proportion of LDAPSharedCacheSize to LDAPCacheEntries is too small, the old cache entries will not get purged. Hence, the cache hit rate drops rapidly and degrades the system performance. This performance bug requires setting both LDAPSharedCacheSize and LDAPCacheEntries options to specific values to manifest. The fix is to change the default setting of LDAPSharedCacheSize (i.e., 200K) to 500K. In addition, as shown in Figure 3.4, the source code is patched by inserting log statements (i.e., ap_log_error) as extra checks to help developers debug the code. For example, if the system fails to allocate memory for a new entry after the purge, events are recorded and the code returns NULL to the caller.

```c
/*Add more buffer*/
byte* buf_pair_ptr[2];
...*/

/* Acquire the log mutex. */
#define log_mutex_enter_all() do {
    mutex_enter(&log_sys->r_mutex);
    mutex_enter(&log_sys->w_mutex);
} while (0)
...

/* Release the log mutex. */
#define log_mutex_exit_all() do {
    mutex_exit(&log_sys->w_mutex);
    mutex_exit(&log_sys->r_mutex);
} while (0)
```

**Figure 3.3 MySQL Bug #77094**

**Abnormal Termination.** In Apache bug #42829, configuring multiple listening ports (e.g., {Listen 80, Listen 8080}) causes Apache server to hang during a graceful restart. This bug happens when a child process receives a software signal that closes the listening sockets before the process starts polling the sockets using the apr_pollset_poll function. The apr_pollset_poll function is only called when multiple listening ports are present. Since no listening sockets are available (closed by the signal), the calling child process waits indefinitely. Figure 3.5 shows the fix for this bug. In the code patch, the timeout is updated to 10 second to avoid waiting apr_pollset_poll function forever. This bug involves 64 posts over the course of four years to close since opened in 2007.

**Figure 3.4 Apache Bug #46749**

```c
node = (util_cache_node_t *)util_add_alloc(cache,
sizeof(util_cache_node_t));
if (node == NULL) {
    return NULL;
    + ap_log_error("LDAPSharedCacheSize is too small...");
    + if (cache->numentries < cache->fullmark)
    + cache->marktime = apr_time_now();
    + }
    + util_add_cache_purge(cache);
    + if (node == NULL){
    + ap_log_error("Could not allocate memory ...");
    + return NULL;
    + }
```
Slow Autocomplete Feature. In Firefox, the autocomplete in the URL bar is expected to expand URLs immediately as users type. In Firefox bug #415489, a user reports that the autocomplete becomes extremely slow. The bug is due to misconfigurations of browser.urlbar.search.chunkSize and browser.urlbar.search.timeout preferences, where the first option defines the number of chunks must be collected from SQLite database, and the second option defines the search timeout value. When the chunk size is set too small, the data query completes fast and waits for the search timeout to return and thus leading to performance degradation. The fix is to balance the chunk size and the timeout value (Figure 3.6).

```
for (;;) {
  status = apr_pollset_poll(pollset,
    -1, ...); /* wait forever*/
  if (APR_STATUS_IS_TIMEUP(status))
    continue;
}
```

Figure 3.5  Apache Bug #42829

The above six bugs can help motivate this study and answer some earlier questions about configuration-related performance bugs in the following ways: 1) Performance bugs are triggered by not only inputs, but also specific configurations. 2) These bugs have shown similarities to general configuration bugs. For example, changing values of configuration options can affect the system performance. Performance issues may also require more than one configuration option to manifest. On the other hand, these bugs are also different from general configuration bugs. The configuration options are more related to memory (Figure 3.4), synchronization (Figures 3.2 and 3.3), and network (Figures 3.1 and 3.5). 3) Existing configuration testing or performance testing may not be effective to handle these bugs. For example, performance testing in the absence of configurations may miss bugs in all six examples. On the other hand, when applying traditional configuration-aware techniques, sampling large configuration space may end up choosing options that are not related to performance.

3.3 Case Study

Our study has two main objectives. First, we intend to understand the complexity of performance bugs in highly-configurable systems. Second, we want to understand what are the
challenges that we will face as we apply configuration-aware techniques to performance issues. Therefore, we consider the following research questions.

RQ1: How prevalent are performance bugs related to configurations?

RQ2: What are the types of configurations that can influence performance?

RQ3: What are the causes of configuration-related performance bugs?

RQ4: How complicated is it to fix configuration-related performance bugs?

3.3.1 Data Sets

We chose three large-scale open-source software projects: Apache, MySQL, and Firefox. With millions of lines of publicly accessible code and well maintained bug repositories, these subjects have been widely used by existing bug characteristic studies [72, 160, 163].

3.3.1.1 Studied Subjects

The selected programs are listed in Column 1 of Table 3.1. The subject programs cover various application spectrums: an HTTP web server, a database engine, and a web browser. All three projects started in the early 2000’s and each has over ten years of bug reports.

Table 3.1 Subjects and Characteristics

<table>
<thead>
<tr>
<th>Application</th>
<th>Sampled</th>
<th>Used</th>
<th>CB</th>
<th>Opt.</th>
<th>UsedOpts.</th>
<th>Configuration Options Appeared More Than Once</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>323</td>
<td>63</td>
<td>60</td>
<td>1,145</td>
<td>35</td>
<td>KeepAlive (5) MaxClients (3) Listen (3) ThreadsPerChild (2) RequestHeader (2) ProxyPass (2) BalencerMember (2) SSLVerifyClient (2) AuthLDAPURL (2)</td>
</tr>
<tr>
<td>MySQL</td>
<td>241</td>
<td>77</td>
<td>41</td>
<td>1,429</td>
<td>31</td>
<td>query_cache_size (4) innodb_flush_log_at_trx_commit (3) innodb_flush_method(2) innodb_thread_concurrency (2) innodb_buffer_pool_size (2) read_buffer_size (2) sort_buffer_size (2) gtid_mode (2) query_cache_type (2) profiling (2) sync_binlog (2)</td>
</tr>
<tr>
<td>Firefox</td>
<td>323</td>
<td>53</td>
<td>12</td>
<td>1,650</td>
<td>24</td>
<td>Places (5) browser.urlbar.search.chunkSize (2) browser.urlbar.search.timeout (2) Collusion (2) browser.urlbar.maxRichResults (2) flash plugin (2)</td>
</tr>
<tr>
<td>Total</td>
<td>887</td>
<td>193</td>
<td>113</td>
<td>4,224</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>

3.3.1.2 Data Collection

Bugs. We picked configuration-related performance bugs from two sources: bug repositories and changelogs. We searched bug databases using a set of performance-related
keywords ("slow", “performance”, “latency”, “throughput”, etc.). We filtered out unconfirmed reports, which yielded a total of 887 bugs (Column 2 of Table 3.1). During the manual inspection, we follow those reports that have sufficient details in bug descriptions and discussions posted by commentators, and decide if the inspected bug is a performance bug or not, and whether the bug is related to configuration or not. We collected both misconfiguration performance bugs and bugs that are triggered by specific configurations.

To ensure the correctness of our results, the manual inspections were performed independently by three graduate students. For bugs where the results from three inspections differ, the authors and the inspectors discussed to reach a consensus. As such, the examination yielded a total of 193 performance bugs, and 113 of them are related to configurations (Column 3-4 in Table 3.1).

Configurations. To answer our research questions, we also need to know the configuration space for each subject. We collected such information by studying all artifacts that are publicly available to users, including documents (e.g., user manuals and online help pages), configuration files, and source code. In Firefox, we also utilized the APIs that have been provided to programmatically manipulate internal data structures that hold configuration information, as well as studied the about:config page (a utility for modifying configurations). This process yielded the total number of configuration options for the three subjects (Column 5 of Table 3.1). Column 6 of Table 3.1 lists the total numbers of configuration options for the 113 studied bugs. The last column lists the configuration options and the number of bugs that each option is associated with (indicated in the parenthesis). We list only options that appeared in more than one studied bugs. These options will be discussed in the next section.

3.3.2 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our subjects and bugs. Other subjects may exhibit different behaviors. Data recorded in bug tracking systems and code version histories can have a systematic bias relative to the full population of bug fixes [21] and can be incomplete or incorrect [12]. However, we do reduce this threat to some extent by using several varieties of well-studied open source code subjects and bug sources for our study.

The primary threat to internal validity involves the use of keyword search and manual inspection to identify the configuration-related performance bugs. The precision of this approach is 100%, which is the percentage of true performance bugs among the performance bugs manually verified by us. To minimize the risk of incorrect results given by manual inspection, bug were labeled as performance bugs independently by three people. The recall of our approach is estimated to be 50%, which means that for each analyzed performance bug,
there is a performance bug that we missed. To compute this recall, we randomly sampled 227 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. [92]. 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 was successfully used in prior studies [72, 92, 160].

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 tracking systems of the three 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 and the number of lines in the patch, which are straightforward to compute.

3.4 Results

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

3.4.1 RQ1: Prevalence of Bugs

To answer RQ1, we turn to Figure 3.7. A total of 193 performance bugs are classified into configuration and non-configuration bugs (solid grey area), where configuration bugs are further classified into three categories (described in RQ2). For the 80 non-configuration performance bugs, defects are normally caused by semantic bugs in code that are independent of configurations. For instance, in MySQL bug #15935, an update query takes a very long time to complete. The degradation of performance is caused by the wasted search of the entire table indices, though only a smaller range of search is necessary. While the defect numbers can be potentially influenced by our sampling strategy, among all four categories, configuration-related performance issues contribute to 59% of the studied cases.

Finding 1: A significant percentage of performance bugs are related to configuration issues (59%). Compared to general bugs (27%) from a previous study [160], if these results generalize, performance bugs are more relevant to configurations.
3.4.2 RQ2: Configuration Types

We begin by answering RQ2 by classifying the configurations of the examined 113 bugs into three general categories: parameter configurations, hidden configurations, and system-specific configurations. The parameter configurations refer to the configurations whose values can be changed by end users through configuration files. The hidden configurations refer to program variables embedded in the source code which can be set only if developers are aware of them. The system-specific configurations refer to configurations related to hardware, system topology, and the choice of system core libraries. Table 3.2 indicates the categories and the total number of bugs falling into each category.

<table>
<thead>
<tr>
<th>Application</th>
<th>Parameter</th>
<th>Hidden</th>
<th>Sys-spec</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>50</td>
<td>1</td>
<td>9</td>
<td>60</td>
</tr>
<tr>
<td>MySQL</td>
<td>32</td>
<td>2</td>
<td>7</td>
<td>41</td>
</tr>
<tr>
<td>Firefox</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

**Finding 2:** The parameter configurations account for a majority of the examined configurations (78% to 92%). This ratio is similar to the finding by Yi et al. [160], where up to 85.5% of general bugs are related parameter configurations.

**Finding 3:** However, the system-specific configurations account for a non-trivial portion (8% to 17%), and should be of particular concern for performance.

We next describe the three configuration categories in more detail.

3.4.2.1 Parameter Configurations
Given the prevalence of parameter configuration bugs, we have studied them from three perspectives: types of configurations, the number of parameters, and the problem domains.

**Types of Parameters.** We look at illegal and legal configurations. The illegal configurations refer to violations of configuration rules related to format, syntax, or semantics. Such configurations are unacceptable to the examined system. For example, a directory name option `datadir` can point to the wrong directory causing MySQL not to start (MySQL bug #65022). However, we have not found any performance bugs related to illegal configurations. Our further investigation exhibits two reasons. First, illegal configurations often lead to functional bugs, such as startup failures and error messages. Second, long term applications usually have established built-in syntax checkers as the first line of defense to guarantee the correctness of configuration files. For instance, after issuing a graceful restart in Apache server, it validates the changed configuration file before booting up.

On the other hand, legal configuration options can still cause performance issues. For instance, some performance bugs, although not directly caused by misconfigurations, require certain configuration options to manifest, such as the example in Figure 3.1. Others can be fixed by updating the configuration values as seen in Figure 3.3 and Figure 3.4. Performance bugs like these are difficult to detect as it requires users’ experience to select the right configurations to expose them. To address this problem, it is possible to leverage some code-level analysis techniques [18, 165].

**Finding 4:** All studied parameter configuration bugs results from legal options. The ratio of the number of bugs caused by legal configuration values over all the studied configuration performance bugs is much higher than the finding (46.3% - 61.9%) for general configuration bugs by Yi et al. [160].

**Numbers of Incorrect Parameters.** Existing configuration-aware techniques mostly focus on single and two configuration options (e.g., 2-way combinatorial testing [32]). There have also been many debugging techniques that focus on only single configuration options [165]. In this study, we examine the percentage of performance bugs involved in different numbers of configuration options – one option, two options and more than two options.

Columns 1-3 in Table 3.3 show the number of parameters involved in configurations that lead to performance bugs. Our analysis indicates that about 27% to 28% of the parameter configuration bugs involve multiple options. For instance, the cache purge example in Figure 3.4 shows a bug case where two options are involved.

**Finding 5:** The majority (72% to 73%) of studied parameter configuration bugs is related to only one option, whereas about 27% to 28% of the examined parameter configuration bugs involve two and more options.
Problem Domains of Configurations. We also study under which problem domain each performance bug falls. We propose five domains based on the functionality of the involved parameters: memory, network, I/O, concurrency, and graphics.

Table 3.3 Parameter Configurations

<table>
<thead>
<tr>
<th>Application</th>
<th>One option</th>
<th>Two options</th>
<th>&gt; 2 options</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>36</td>
<td>11</td>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>MySQL</td>
<td>23</td>
<td>7</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>Firefox</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 3.4 indicates the domains and the total number of bugs falling into each category. Other studied bugs do not fall into these categories hence purposefully left out. For instance, the example in Figure 3.4 (i.e., LDAPSharedCacheSize) falls into the memory domain. The example in Figure 3.5 (i.e., Listen) falls into the network domain. In fact, 87% of parameter configurations in Apache fall into the Network domain, and 67% of parameter configurations in MySQL are equally distributed to Memory and Network domains.

Table 3.4 Types of Configurations

<table>
<thead>
<tr>
<th>Application</th>
<th>Memory</th>
<th>Network</th>
<th>I/O</th>
<th>Concurrency</th>
<th>Graphics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>2</td>
<td>27</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>12</td>
<td>12</td>
<td>8</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Firefox</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>15</td>
<td>39</td>
<td>9</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

The concurrency domain involves options that manipulate threads and synchronizations (e.g., the skip-thread-priority in MySQL bug #37536). The I/O domain involves options that affect I/O operations. For example, in MySQL bug #61818, performance can be adjusted by setting the option innodb_flush_log_at_trx_commit which controls the timing of flushing log files to disks. The graphics domain involves the control of the display. For instance, in Firefox bug #820247, a noticeable lagging occurs when rendering the Bookmarks submenu. Graphics rendering is much faster in this bug after turning off the Hardware Acceleration (GPU) in the preferences (“Use hardware acceleration when available”).

Finding 6: The majority of the examined parameters fall into the Memory and Network domains. However, the distribution depends on the characteristics of applications (e.g., Firefox is more relevant to the graphics domain).

Table 3.5 System-Specific Configurations

<table>
<thead>
<tr>
<th>Application</th>
<th>Environment</th>
<th>HW</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>1</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>MySQL</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Firefox</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
3.4.2.2 Hidden Configurations

There are hidden configurations found only in the source code and related files. These hidden configurations are difficult to handle by existing techniques such as reverse engineering [112], since many applications are written in multiple programming languages such as C++, Java, and JavaScript, and often use aliases to refer to configuration option names. In both cases, they are not supported by existing techniques. Jin et al. [71] has reported that up to 44% options are hidden in the code and user manuals.

However, our observation is that only a small portion (three cases) of hidden configurations in the examined parameter configuration bugs are related to performance bugs. All three configurations are found in the source code. The first case, MySQL bug #20876, shows the default value of FILE_SYSTEM_HASH_SIZE is set to a small value, causing the CPU to spike to 100%. The second case found in MySQL bug #64258 concerns the default read timeout value (WAIT_FOR_READ) for storage devices, which is set to be unnecessarily long. The last one is in Apache bug #58091, where MC_DEFAULT_SERVER_TTL is set too short for the connection timeout.

**Finding 7:** A small portion (3%) of the examined performance configuration bugs involve hidden options.

3.4.2.3 System-Level Configurations

The system-specific configurations refer to the misconfigurations of the external environment under which an application executes, such as system topology (environment setup), hardware choice, and incompatible components/libraries. Table 3.5 shows the three types and the total number of bugs falling into each type.

While a complex environment setup serves its purposes, it could also be a source of performance issues, as we found in the Apache bug #45834. Apache is used to host SVN on a Red Hat server with a firewall that sits in between the hosting server and the LDAP server. The authentication process (mod_authnz_ldap) takes a long time to complete without doing any useful work. The root cause of this bug is that the firewall breaks the established connection between the SVN server and the LDAP server, and hence causes the retransmitting of TCP packets. Changing the timeout value in the LDAP component fixes the bug.
Another cause of performance bugs involve choosing inappropriate hardware. This can happen when customers choose between different hardware settings. For instance, some Intel CPU models equipped with Hyper-Threading (HT) technology [66] are supposed to boost throughput on threaded applications. Nonetheless, from MySQL bug #15815 we have learned that concurrency performance is much worse when HT is enabled due to lock contentions.

Software incompatibility is another major cause of performance bugs. The application may work well under a standard set of components, yet the performance problems can occur when the components are improperly customized. For example, in the Apache bug #40010, the server becomes unresponsive on a FreeBSD OS with Jails installed. The reason is that with Jails installed, the IP address used by the Apache server is not mapped to the appropriate IPv6 address on FreeBSD, causing the connection to be rejected. The rejected connection freezes the server.

**Finding 8:** There are various system-level misconfigurations that can cause performance bugs. The majority (70%) of the examined system-specific configurations involve incompatible components/libraries.

3.4.3 RQ3: Causes of Bugs

We investigate the causes of performance bugs by looking at when and why performance bugs were introduced to the system.

3.4.3.1 When are Bugs Introduced?

We examine the stages where configuration-related performance bugs are introduced. We categorize these cases into (1) first-time use, (2) runtime reconfiguration, and (3) application upgrade. Table 3.6 shows the three stages and the total number of bugs falling into each category.

We define the first-time use as configuration files initially read from persistent storage (e.g., hard disk) into memory. The causes for the configuration bugs during first-time use can be inadequate domain knowledge, design defects of the system, user mistakes, or even inconsistent manuals [114]. In the case of runtime reconfiguration, when applications are running, a user can modify the configuration files directly. Only in Firefox will a modification take effect immediately and be written back to the preference files. The configuration options browser.urlbar.search.chunkSize, and browser.urlbar.search.timeout in the example of Figure 3.6 can be reconfigured at runtime. In contrast, in Apache and MySQL, the dynamic memory is not updated. The changed configurations are held in temporary memory and will take effect at the next startup.
Table 3.6 Stages of Bugs

<table>
<thead>
<tr>
<th>Application</th>
<th>First-time</th>
<th>Runtime</th>
<th>Upgrade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>60</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>MySQL</td>
<td>37</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Firefox</td>
<td>11</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>108</td>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.7 Causes of Bugs

<table>
<thead>
<tr>
<th>Application</th>
<th>Memory</th>
<th>CPU</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>24</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>MySQL</td>
<td>13</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>Firefox</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Performance bugs can also be introduced when upgrading an application even if there are no changes in the configuration files. For example, in Apache bug #58037, after Apache server updates from version 2.2 to 2.4, the time to complete authentication grows as the number of authentication connection increases, though all configuration values remain unchanged. This bug is due to the LDAPConnectionPoolTTL option, whose default value is zero in the old version. However, in the new version, the server only unlocks connection when this option is not set to zero. As a result, the size of the connection entries grows rapidly. By the time a new connection is issued, all the locked connections will be checked, the time spent on checking connections that should have been unlocked explains the increased processing time. This bug can be avoided by setting the LDAPConnectionPoolTTL option to a non-zero value.

**Finding 9:** The majority (95%) of studied performance bugs are related to the first-time use. However, in applications that allow runtime reconfiguration, the performance bugs caused by runtime misconfiguration are non-negligible (11% in Firefox).

3.4.3.2 Why do Performance Bugs Happen?

We further examine the causes of performance bugs based on observable internal symptoms. In contrast to external symptoms, where the application simply hangs or slows down, the internal symptoms reflect the possible presence of performance bugs. We consider three major internal symptoms: 1) memory usage, 2) CPU usage, and 3) synchronization. These symptoms can be observed by profiling and system monitoring tools. Table 3.7 indicates the three categories, the number of bugs, and the configuration domains falling into each category.

The memory problem can refer to memory leak where the program fails to release memory when no longer needed, or low cache hits. The inefficient memory usage can negatively impact the performance of the systems [157]. For instance, in Apache bug #44975, when both mod_ssl
and mod_deflate are enabled, a per connection memory leak is triggered by the client who initiates an SSL handshake with compression algorithm support. Apache bug #46749 in Figure 3.4 shows the symptom of low cache hits. The high CPU usage means that the running programs hold a lot of CPU resources, which is an indicator of performance anomalies [149]. In the MySQL bug #55629, with a large read buffer (read_buffer_size), the wrong error code returned during I/O cache flushing from a select query causes the database server to loop indefinitely and results a high CPU usage.

The inefficient synchronization usage refers to the scenario when multiple threads contend to reach a synchronization point. Such synchronization bottlenecks can lead to performance bugs. For example, in MySQL bug #37536, the skip-thread-priority option is used to change thread priorities. However, this option can cause significant performance degradation depending on the thread count and number of processors.

**Finding 10:** A majority of performance bugs involves inefficient memory usage (up to 35%), and a significant portion of performance bugs are caused by high CPU utilization (up to 17%) and synchronization (up to 14%).

<table>
<thead>
<tr>
<th>Application</th>
<th>Options</th>
<th>Source</th>
<th>Option &amp; Source</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>5</td>
<td>54</td>
<td>1</td>
<td>60</td>
</tr>
<tr>
<td>MySQL</td>
<td>2</td>
<td>39</td>
<td>0</td>
<td>41</td>
</tr>
<tr>
<td>Firefox</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

3.4.3.3 What configuration options are culprits?

As the last column in Table 3.1 shows, in all studied bugs, some configuration options appear more than once (nine in Apache, 11 in MySQL, and six in Firefox). We conjecture that these options are more likely to relate to performance bugs. The anecdotal evidence from the recent release of MySQL [133] also indicates that only 17 configuration options are the most common causes for MySQL performance degradation. The 17 options are also covered by the 31 configuration options studied in MySQL.

**Finding 11:** Performance bugs are caused by a small subset of configuration options.

3.4.4 RQ4: Bug Fixes

Finally, we examine the fix of performance bugs in highly-configurable systems. We focus on how to fix the bugs and the complexity of the fixes.
3.4.4.1 How to Fix Performance Bugs?

Fixing performance bugs can be done by (1) changing configuration values, (2) patching source code, and (3) fixing both configuration values and source code. Table 3.8 shows the three categories and the number of bugs that fall into each category.

We find that only a few cases require fixing solely configuration options. In contrast, 88% of performance bugs need to be fixed at the code level, such as the examples in Figures 3.1, 3.3, and 3.5. About 2% of performance bugs require fixing both configurations and the code. The example in Figure 3.4 is such a case.

**Finding 12:** A dominate majority (88%) of performance bugs involve fixing the code.

3.4.4.2 Are Patches Complex?

Jin et al. [72] study general performance bugs and their results indicate that performance bugs can be fixed by simple patches (e.g., eight lines on average). We find that fixing performance bugs in highly-configurable systems is not that simple. In fact, up to 61% of studied bugs require fixing over 30 lines of code.

**Finding 13:** Fixing configuration-related performance bugs is more complex than fixing general performance bugs.

3.5 Discussion

Our study motivates further research on performance testing – performance bugs cannot be exposed easily without specific configurations. 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 in highly-configurable applications.

3.5.1 Implications to Practitioners

**Performance testing should consider key configurations.** Configuration-aware testing has been widely adopted in industrial environment [109] (e.g., combinatorial interaction testing [6]). Configuration-aware testing can be expensive because the space of possible unique configuration combinations grows exponentially with the configuration options. To address this
problem, testers often evaluate a representative sample of all possible configurations [111, 159].

Given a large configuration space, certain configuration options are more likely to trigger performance bugs than the others (Finding 11). In contrast to the standard configuration-aware testing [111], when doing performance testing, developers can prioritize configuration options that are more relevant to performance. Our results also suggest that performance testing can focus on one or two configuration options (Finding 5).

Although source code is the ground truth [71] of the configuration space, it may not be available to developers who want to test hidden configurations. Fortunately, our results indicate that only a small portion of configuration options are hidden options, and hence it is less likely to have caused performance issues (Finding 7).

System-level configurations are important. As our results have shown, a significant portion of performance bugs are related to system-specific configurations (Finding 3). This implies that developers need to test their applications under different system settings that can closely resemble production. In addition, when a performance bug occurs, developers should not limit their searching efforts restricted by the target application scope but the underlying executing environment as well.

Profiling is helpful for identifying misconfigurations. Profiling is frequently used to bootstrap performance diagnosis and allows developers to collect various system statistics (internal symptoms) such as CPU utilization, cache hit/miss rate and synchronization time. Given the results of our study, we have provided insights about linking each configuration domain with its internal symptoms (see Table 3.7). Such information could be of help to pinpoint configuration options when using profilers to observe anomalous system statistics (e.g., the low cache hit rate may be associated with the configuration options in the memory domain).

3.5.2 Implications to Researchers

Testing and Debugging Tools Are Needed. As we have seen, the current state of research in testing for performance bugs consider two major aspects – test inputs and test oracles [93, 107]. Yet this is not realistic for highly-configurable applications. We have seen in many cases, exposing bugs require both specific inputs and configuration options. We need, therefore, new configuration-aware techniques to test for and debug performance bugs.

One possibility is to leverage existing static analyses [81, 112] to identify performance-sensitive configuration options based on code patterns. Such options can be used to guide performance testing. In the example of Figure 3.4, the configuration options related to
util_ald_alloc are performance-sensitive. Therefore, effective techniques should be developed to unify the mapping of configuration items back to the code base.

To facilitate performance debugging and diagnosis, we need to link the erroneous behavior to the buggy code, inputs and configuration options. Most existing performance debugging techniques assume that inputs and configurations are available and only identify buggy code that leads to performance issues [131]. However, when a performance bug occurs, users may not even think of configuration as a cause of their problems. While existing configuration debugging and diagnosis techniques [18, 165] can address general functional bugs, real-world performance bugs are more difficult to handle [72, 131]. We need, therefore, new performance debugging techniques that can isolate bugs caused by misconfigurations and link the bug to specific configuration options.

**Configuration-aware regression testing is needed.** As software evolves, new performance bugs can be introduced (e.g., the Apache bug #58037 in Section 3.4.3). Regression testing is used to perform re-validation of evolving software. To date, most regression testing research has focused on selecting, reducing, prioritizing, and augmenting test inputs [161] while treating software systems as if they possessed a single homogeneous configuration. In addition, existing regression testing techniques do not target performance bugs. It is possible that we can first apply static impact analysis to identify configurations that are affected by performance-sensitive code changes. We could then apply regression testing techniques to detect performance regressions.

**Building performance-influential configuration models.** A performance-influential model can be used to describe how configuration options and their interactions influence the performance of a software system. There has been a great deal of research on building such models to help developers predict performance [59, 78, 126] through various techniques such as sampling and machine learning. We believe that these black-box techniques can be improved if performance-sensitive configuration options and their associated code elements can be better understood. First, in a system with enormous configuration options, minimizing learning set is still a challenge. Identifying key configuration options will mitigate such problems. Second, black-box techniques only approximate the performance influence induced by various combinations of configuration options. As such, more sophisticated techniques are needed to analyze these options, such as the dependency analysis. Finally, system-specific configuration options can be used to model environment for building performance models.

**Fixing and avoiding performance bugs.** Many software systems require continued operation even if an erroneous condition is met. For example, self-adaptive software systems provide adaptation mechanisms that allow continued operation when the system environment changes. In the case of performance bugs, a self-adaptive system should be able to reconfigure
its settings to meet the performance requirements. If we return to our results of the study, the performance bugs always show certain internal symptoms such as high CPU utilization and low cache hits (Finding 10). Monitors can be used to capture such information, so that the application is reconfigured to certain settings if a performance bug symptom manifests. Research has shown that workarounds can be found to adjust the runtime configurations [137], so it is possible that we can leverage some of those ideas for this work.

**Extracting new configurations.** We have seen two cases where the bug fix involves introducing new configuration options, such as the example in Figure 3.2. While current reverse engineering techniques can extract existing configuration options from the code [112], they cannot infer new configuration options. In order to extract performance-influential configuration options, we need to understand how a certain performance-related feature has been implemented. It is possible that we could leverage ideas of feature localization [45] to derive correspondences between configuration options and computational units, yet this may also yield unnecessary options such that changing their values will not impact performance. Therefore, we need new techniques that can both infer performance features from the code, and determine which features are useful as configuration options.

### 3.6 Conclusion

We have performed a comprehensive characteristic study on 113 configuration-related performance bugs collected from three popular open source projects. With the increasing significance of performance bugs, this activity provides the first study on real-world performance bugs in highly-configurable software systems. Our study covers a wide spectrum of characteristics, including types, causes, symptoms, and fixes. The study provides guidance for future research on performance testing and debugging. We intend to help developers and practitioners to extend and improve tools that can address performance issues in highly-configurable applications. This work is only a starting point for understanding performance bugs related to configurations. In the future, we will extend our study on more subject programs and propose techniques to handle these bugs.
Performance is one of the key aspects of non-functional qualities, where 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. However, researchers often spend a considerable amount of time and effort in finding usable performance bugs because they are difficult to reproduce. In this chapter, we study the characteristics of reported performance bugs by reproducing them in our system environment to examine the challenges of bug reproduction from the perspective of researchers. We spent more than 800 hours over the course of six months to study and 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 from 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.

4.1 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 [17] which leads to problems such as poor user experience, long response time, and low system throughput [26, 62, 72, 92, 149]. For instance, performance bugs have occurred on well-tested software such as the Internet Explorer installed on Windows systems [62], and have caused severe damages to the user experience.

Compared to functional bugs, performance bugs are substantially more difficult to handle [17, 38] because they often manifest themselves through large inputs and specific execution environments [92, 96]. 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 [27, 62, 72, 73, 93, 96]. Although these techniques can detect performance bugs in 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 [26], Github Issue Tracker [52]) 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. Based on the description of a solved performance bug report, researchers can determine whether the performance bug can be used in their evaluation. A failed-to-reproduce performance bug is discarded when it cannot be reproduced by researchers, often due to lack of domain knowledge or environment limitations (e.g., compilation, dependencies, etc.). Therefore, the bug selection 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 [38] 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 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 researchers, rather than understanding non-reproducible bugs from the viewpoints of developers. Therefore, we focus on studying bug reports that have already been resolved by developers. 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 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.

The rest of the paper is organized as follows. We first present motivating examples in Section 4.2. We then describe our methodology for choosing subject applications, the bugs selected to study, and the threats to validity in Section 4.3. Our results are demonstrated in Section 4.4, followed by discussions in Section 4.5. We present related work in Section 4.6 and end with conclusions in Section 4.7.
4.2 Motivating Examples

We refer to software performance bugs as programming errors [92, 93] and configuration errors that cause significant performance degradation. They can adversely affect speed, throughput, and responsiveness of the system, which leads to the poor user experience. Some other terms such as “performance problem” and “performance issue” are also widely used [92]. In this paper, we use these terms interchangeably.

We use three examples from 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 of success in performance bug reproduction? The three bug reports represent three difficulty levels of reproducing performance bugs: from difficult (“failed-to-reproduce”) to medium (“reproducible with effort”) to easy (“reproducible”).

A Failed-To-Reproduce Performance Bug Report: 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 was not mentioned in the bug report. Since there are 34 releases under Apache 2.2, on average, compiling and installing Apache from source code can take anywhere between 20 to 50 minutes; it is too time-consuming (up to 28 hours 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 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.

A Reproducible Performance Bug Report with Effort: 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 reconfigured 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 hours to research on plausible components to fill in the missing information and finally reproduced 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 0.9.7a). To observe the performance bug symptom, we use the Apache benchmark ab to request 10,000 pages with 50 threads enabled: “ab -n 10000 -c 50 https://localhost:443/”.

A Reproducible Performance Bug Report: MySQL Bug #74325. This performance regression bug happens in MySQL version 5.7.5. When compared to MySQL v5.0.85, MySQL v5.7.5 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 than 5.6.21” 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 five hours to successfully reproduce the bug.
4.3 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?

4.3.1 Data Sets

4.3.1.1 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 [72, 160, 163]. The selected programs are listed in Column 1 of Table 4.1. Both projects started in the early 2000s and each has over ten years of bug reports.

4.3.1.2 Data Collection

We collected performance bugs from the bug system of Apache [14] and MySQL [88]. 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 [27, 62, 72, 73, 93, 96]. 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 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 five, 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 4.1 list the release versions of the subjects and performance bugs selected from each version.

4.3.1.3 Study Setup

We build the environment as a virtual disk image (VDI) on the VirtualBox [146] 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 to 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, 16 GB of memory, and 512 GB 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, 4 GB of memory, and 120 GB 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, 36 GB memory, and 256 GB 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 hours 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 is not present in the bug report. From the perspective of researchers who try to reproduce a bug directly following instructions available in the bug report, such bugs are considered failed-to-reproduce.

**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.

**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) [86]. 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.

4.3.2 Threats to Validity

The primary threat to 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 [21] and can be
incomplete or incorrect [12]. 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 internal validity 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.

4.4 Results

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

4.4.1 RQ1: Reproduced Bug Reports and Their Characteristics

Column #Rep and #Failed of Table 4.1 list the number of reproduced and failed-to-reproduce bugs across different versions of the two subjects.

Table 4.1 Subject Characteristics

<table>
<thead>
<tr>
<th>Subject</th>
<th>Init Rel</th>
<th>Last Rel</th>
<th>#Sampled</th>
<th>#Failed</th>
<th>#Rep</th>
<th>Success Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache 2.0</td>
<td>2002</td>
<td>2013</td>
<td>20</td>
<td>16</td>
<td>4</td>
<td>20%</td>
</tr>
<tr>
<td>Apache 2.2</td>
<td>2005</td>
<td>2017</td>
<td>31</td>
<td>26</td>
<td>5</td>
<td>16%</td>
</tr>
</tbody>
</table>
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 4.2. The characteristics of the failed-to-reproduce bug reports will be discussed in Section 4.4.2.

Table 4.2 Reproduced Bugs and Their Characteristics

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

When reproducing a server-side performance bug, environment setup, data inputs, configuration options, and input actions are four essential elements. Figure 4.1 shows an example of the use of the four elements in Apache bug #48024. Environment setup refers to steps to install OS and set up specific application components (e.g. in Figure 4.1, 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 Figure 4.1 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 [140]. 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 Figure 4.1 we have included the ProxyPass option) correspond to the customizable items in the configuration file. Input action refers to the logical steps to take after environment setup for the performance bug to manifest (e.g. in Figure 4.1, the action includes issuing an HTTP request).

<table>
<thead>
<tr>
<th>Environment Setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. export INSTALLED=/PWD/apache-install/</td>
</tr>
<tr>
<td>2. ./configure --prefix=$INSTALL --enable-sed --enable-proxy</td>
</tr>
<tr>
<td>12. python -ms SimpleHTTPServer &amp; #Start backend server</td>
</tr>
<tr>
<td>13. /bin/apachectl start #Start proxy server</td>
</tr>
<tr>
<td>Data Input</td>
</tr>
<tr>
<td>A static file with 1M + characters on a single line</td>
</tr>
<tr>
<td>Configuration Options</td>
</tr>
<tr>
<td>Header unset Content-Length</td>
</tr>
<tr>
<td>SetOutputFilter Sed</td>
</tr>
<tr>
<td>ProxyPass / <a href="http://127.0.0.1:8000/">http://127.0.0.1:8000/</a></td>
</tr>
<tr>
<td>Input Actions</td>
</tr>
<tr>
<td>HTTP request: <a href="http://localhost:8000/a.1">http://localhost:8000/a.1</a></td>
</tr>
</tbody>
</table>

Figure 4.1 Reproducing Apache Bug #48024

Column Set of Table 4.2 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. Figure 4.1 shows part of the 13 steps of environment setup for reproducing Apache bug #48024.

Columns Inp of Table 4.2 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 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 Figure 4.1. 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 4.2 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 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 three 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 4.2 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 4.3 summarizes the types of workloads in the 17 reproduced bug reports, including network traffic and database operations.

<table>
<thead>
<tr>
<th>Workload Type</th>
<th>Description</th>
<th>Bug Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Traffic</td>
<td>Concurrent web page requests</td>
<td>Apache bug #54852</td>
</tr>
<tr>
<td>Web Traffic</td>
<td>Long HTTP connection sessions</td>
<td>Apache bug #43081</td>
</tr>
<tr>
<td>MySQL</td>
<td>Large number of database tables</td>
<td>MySQL bug #15653</td>
</tr>
<tr>
<td>MySQL</td>
<td>Concurrent updates on DB tables</td>
<td>MySQL bug #74325</td>
</tr>
</tbody>
</table>

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

Column Act of Table 4.2 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 Figure 4.1, sending an HTTP request is an action.

**Finding 6:** A majority (88%) of the reproduced performance bug reports require no more than three input actions.
Column Order of Table 4.2 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 Figure 4.2.

```
1. Replace default port with "Listen 50000"
2. Add another port option "Listen 50001"
3. Restart server
4. Make a request on port 50000
5. Delete option "Listen 50001"
6. Restart server
7. Make a request on port 50000
```

Figure 4.2 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 4.2 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 five seconds. Afterward, the CPU usage level returns to a normal state.

Column Workaround of Table 4.2 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.

4.4.2 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 Figure 4.3.
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%).

**Table 4.4 Performance Bugs Failed-To-Reproduce**

<table>
<thead>
<tr>
<th>Problem</th>
<th>Subject</th>
<th>BugID</th>
<th>Bug Description &amp; Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Dependency</td>
<td>MySQL</td>
<td>61188</td>
<td>Slow performance on dropping compressed tables. The bug requires 20 GB of memory to manifest.</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>64258</td>
<td>High read timeout on InnoDB engine causes longer mutex wait. Lack of SSD on dev environment.</td>
</tr>
<tr>
<td></td>
<td>Apache</td>
<td>24448</td>
<td>Java applet consumes significant CPU while Apache used as a proxy. Bug requires a hardware device to host backend server.</td>
</tr>
<tr>
<td>OS Dependency</td>
<td>MySQL</td>
<td>52102</td>
<td>InnoDB plugs has worse performance than built-in InnoDB engine. Bug requires Microsoft Windows.</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>18526</td>
<td>Thread priority is enabled by default on OS X which lowers performance. Bug requires OS X.</td>
</tr>
<tr>
<td>Component Dependency</td>
<td>Apache</td>
<td>38602</td>
<td>Web server does not keep HTTP connections alive. JBoss v3.2 is not available.</td>
</tr>
<tr>
<td></td>
<td>Apache</td>
<td>45834</td>
<td>Authentication takes up to 15 mins to finish with mo_authnzIdap module. The firewall that sits between servers is unknown.</td>
</tr>
<tr>
<td>Source Code</td>
<td>MySQL</td>
<td>26079</td>
<td>Database hangs during binlog rotation when InnoDB engine is</td>
</tr>
<tr>
<td>Issue Type</td>
<td>Product</td>
<td>Issue ID</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------</td>
<td>----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Unavailability</td>
<td>MySQL</td>
<td>30414</td>
<td>Performance regression in throughput tests when logging is enabled. MySQL version 5.1.21 and 5.1.20 are not available.</td>
</tr>
<tr>
<td></td>
<td>Apache</td>
<td>35686</td>
<td>Memory leak due to the multi-threaded MPM worker module. Apache fails to build with OpenSSL.</td>
</tr>
<tr>
<td></td>
<td>Apache</td>
<td>12757</td>
<td>LDAP cache fails to create cache file on all processes except the first one. GCC is not compatible with the source code.</td>
</tr>
<tr>
<td></td>
<td>Apache</td>
<td>38403</td>
<td>Child thread consumes 100% CPU as Apache used as a reverse proxy server. Configure utility failed with an syntax error message.</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>24148</td>
<td>Database hangs when closing SSL connections. MySQL failed to recognize OpenSSL and crashed</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>15811</td>
<td>Long execution time of insert statements with multi-byte character sets. Installation errors out with recipe for target x failed.</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>26527</td>
<td>SQL insertion with LOAD DATA INFILE is very slow in partitioned tables. Error make install failed with message “recipe for target failed”.</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>77094</td>
<td>System log buffer mutex contention. Failed to install the specific sysbench version.</td>
</tr>
<tr>
<td></td>
<td>Apache</td>
<td>45445</td>
<td>The connection timeout causes stalling on unreachable backend servers. Bug requires a busy server with long-lived requests.</td>
</tr>
<tr>
<td></td>
<td>Apache</td>
<td>22106</td>
<td>Embedded SSI slows down web pages. Lack of information on bug reproduction steps.</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>27501</td>
<td>A significant increase in kernel time due to excessive getrusage() calls. Steps to reproduce the bug are very limited.</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>38551</td>
<td>Query cache consumes CPU time even when it is turned off. Lack of instructions on how to trigger the bug.</td>
</tr>
<tr>
<td></td>
<td>Apache</td>
<td>44026</td>
<td>Server memory surges to 16 GB when used as forward proxy. The expected level of memory usage is not observed.</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>15815</td>
<td>Queries take significant longer if multiple queries are running concurrently. Linear time instead of exponential decay is observed.</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>20876</td>
<td>CPU spikes when creating 5k+ tables with large. FIL_SYSTEM_HASH_SIZE. Cannot observe the difference by adjusting option values.</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>39253</td>
<td>Large query cache causes extended blocked mutex wait time. Cannot observe the symptom specified in the bug report.</td>
</tr>
</tbody>
</table>

**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 [8, 87]. 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.

Figure 4.3 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 4.4 describes 24 representatives failed-to-reproduce performance bug reports under each factor.

Table 4.5 Performance Bug Reproduction Problems and Suggestions

<table>
<thead>
<tr>
<th>Problem</th>
<th>Suggestions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Dependency</td>
<td>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.</td>
</tr>
<tr>
<td>OS Dependency</td>
<td>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, version 2.2 can also run on Ubuntu although the bug is originally reported on RedHat.</td>
</tr>
<tr>
<td>Component Dependency</td>
<td>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.</td>
</tr>
<tr>
<td>Source Code Unavailability</td>
<td>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 version 2.4, and by removing the patch from this version, we can reconstruct the faulty version.</td>
</tr>
<tr>
<td>Compilation Error</td>
<td>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.</td>
</tr>
<tr>
<td>Installation Error</td>
<td>Missing files during installation: try to skip deploying non-essential files. For instance, when we install openssl 0.9.7, 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.</td>
</tr>
</tbody>
</table>
| 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. |

4.4.3 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.

4.4.3.1 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 four GB of memory on our machine. We are still able to reproduce this bug as long as we can observe the symptom that all four 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.

4.4.3.2 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.

4.4.3.3 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.

4.4.3.4 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 2.2.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 2.4.x version, and by removing the patched code, we are able to generate a faulty version and reproduce the bug.

4.4.3.5 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.

4.4.3.6 Installation Error

Installation is the last step towards completing the 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.

4.4.3.7 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 “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.

4.4.3.8 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 [9]. 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 4.5 provides a quick reference to the problems of performance bug report reproduction and their solutions. Table 4.6 reports the effectiveness of workarounds applied to the failed-to-reproduce bug reports for the eight failing factors. Column #Failed of Table 4.6 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 because 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.

Table 4.6 Workaround Efficiency and Effectiveness

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

4.5 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.

4.5.1 Implications to Researchers

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. [30] 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 4.4.2), 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.

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 [93, 107] 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.

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 [93, 107]. 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 [111, 159]. One possibility is to leverage existing static analysis [81, 112] 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).

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.

4.5.2 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.

Writing Good Quality Bug Reports is Important. As the last column of Table 4.1 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 4.4.2 (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 [30] 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 4.4.2. 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.

**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 4.6 also suggests that source code unavailability is the easiest to fix, whereas lack of symptom is the most difficult barrier to overcome. Therefore, developers can allocate their efforts to find workarounds according to the causes of the failed-to-reproduce performance bug reports.

4.6 Related Work

**Studies of Bug Reproducibility.** There is a great deal of research on studying the reproducibility of bug reports [30, 34, 47, 50, 53, 57, 58, 118]. Mona et al. [47] mine 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. [118] conduct an empirical study on the characteristics of bugs that influence the reproducibility in the server production environment. They randomly select and inspect 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 propos automated approaches for bug diagnosis. Our study and Sahoo’s work share similarities in that we both study server applications, a set of confirmed bugs, the number of inputs to trigger a bug, and the bug symptoms. Cotroneo et al. [34] conduct a comprehensive study on the characteristics of bug manifestation process. In the study, they identify 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 study 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 focus on reproducing performance bugs, whereas the prior work study 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 does not consider the characteristics that are specific to performance bugs. Second, we focus on the study from the perspective of researchers who try to replicate a known reproducible performance bug with only the description of a bug report. Therefore, we select confirmed performance bug reports that are known to be reproducible by developers, whereas prior work has different target audiences of their studies. Third, prior work study the characteristics of the bugs from bug reports without trying to actually reproduce them in the real environment. In contrast, we get first-hand experience from the perspective of researchers, and go through all the steps necessary to actually execute and reproduce performance bugs, and hence we are able to deliver a reusable set of benchmarks that contain performance bugs.

Chaparro et al. [30] utilize natural language processing and machine learning techniques to automatically identify if bug reports miss important information that can affect understandability and reproducibility. Their work focus on analyzing bug reports and selecting linguistic patterns as machine learning features to automate detection of missing information in a bug report. Our study give 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 et al. [53] classify bugs into Bohrbugs that were easily reproduced with certain inputs and Heisenbugs that are 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 refers to the type of bugs that are complex and non-deterministic. Our studied bugs may fall into the category of Mandelbugs.

Grottke et al. [58] re-define 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 clarify the relationship and definitions for Bohrbugs, Mandelbugs, and Heisenbugs. Later work by Grottke et al. [57] conduct an empirical study in NASA space mission system software. They investigate 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 is independent of characteristics, such as failure effect and failure risk in the space mission system software. The bugs used in our study may fall into the category of aging-related bugs, which is defined as
“faults that can potentially cause software aging, which result in an increased failure rate and degraded performance”.

Frattini et al. [50] discuss the process and influential factors in bug manifestation. Specifically, they survey the taxonomy of bug reproducibility, describe the procedure for manually analyzing a bug report for its reproducibility, and apply machine learning techniques to predict bug classifications. They manually examine if the report is a real bug, and if not, the bug is marked as “NOT_BUG” or “UNKNOWN”. Next, for bugs that have sufficient information, the following is examined: inputs and the application configurations required for exposing the bug. Our manual bug selection approach is similar to theirs as we also utilize the bug repository system to filter out unwanted types of bugs (e.g., the NOT BUG class). We also examine the bugs carefully to identify the inputs and workloads that are required to expose the performance bugs.

There are several differences between Frattini’s work and our study. First, Frattini’s work focus 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 try to actually reproduce the bugs, so we are able to identify more factors influencing reproducibility. We also suggest workarounds to improve the bug reproduction success rate.

Cavezza et al. [29] study 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 demonstrate 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 investigate specific aspects of reproduction (e.g., determinism, environmental factors) for bugs in general, whereas we systematically study a set of fine-grained factors (e.g., input parameters, configurations, reproducing steps) affecting the reproducibility of performance bugs. In addition, we provide 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.

**Performance Bug Empirical Studies.** There has been some work on the empirical study for performance bugs [63, 72, 92, 163]. Jin et al. [72] study 110 performance bugs from five software projects. They study how performance bugs were introduced, exposed, and fixed. They look at the root causes of performance bugs and the code patches. By observing the code patterns that fixed performance bugs, they summarize 25 efficiency rules. They then use these rules to detect performance bugs based on pattern matching. Nistor et al. [92] conduct 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 provide empirical evidence on the importance and challenges of performance bugs. They focus on the way that bugs are discovered and reported, where the authors claim 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 are identified with profilers. They report the complexity involved in the bug fixing and concluded that performance bugs are likely to be more challenging to fix. Zaman et al. [163] study 400 randomly selected performance and non-performance bug reports in Firefox and Chrome. They quantify 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 find that performance bugs are more difficult to handle than nonperformance bugs. Han et al. [63] study the characteristics of 113 performance bugs in highly-configurable systems. They categorize the causes and fixes in performance bugs. A highlight of their study is to point out that configuration options are often neglected in the performance testing although some configuration options can cause performance bugs. While previous research provides insights on identifying the root causes of performance bugs and guidance on addressing performance bugs in general, they do not conduct the study by actually reproducing bugs from performance bug reports.

**Performance Debugging and Testing:** Several techniques in testing, debugging, fixing, and avoiding performance bugs have been proposed in recent literature [54, 62, 73, 93, 107]. Han et al. [62] propose StackMine, a debugging technique to discover high-performance impact call sequences from numerous and complicated call stack traces. Jovic et al. [73] introduce Lag Hunting, a method that monitors deployed interactive system behavior and provides a list of performance issues. The authors argue that the use of profilers would not work for detecting perceptible performance slowness in interactive applications. Instead, they measure the latency to catch perceptible performance problems. Pradel et al. [107] design a regression testing technique to generate performance test cases for thread-safe Java concurrent classes. Grechanik et al. [54] propose a test generation framework, FOREPOST, to associate test inputs with their performance loads. Execution traces are clustered and used to train a classification algorithm to generate rules that describe the semantic patterns of good test inputs. Nistor et al. [93] propose 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 consider only data inputs. Our study acknowledge prior work and suggest that a significant portion of performance bugs are related to configurations, input actions, and the order of input actions. These factors should be considered when designing software testing and diagnosis tools.
4.7 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.
In this chapter, we present CoProf, an approach for performance profiling of configurable software systems that can help developers understand how configuration options and their interactions influence the performance of a system. CoProf combines dynamic program analysis, machine learning, and feedback-directed configuration sampling to profile the program execution and analyze configuration options relevant to performance. In contrast to existing approaches, CoProf uses a white-box approach combined with machine learning to learn performance-influencing configuration options from a few carefully selected executions of the system. We evaluate the approach with 13 scenarios of four real-world, highly-configurable software systems. The results show that CoProf can rank performance-influencing configuration options with high accuracy.

5.1 Introduction

Modern software systems are highly-configurable, allowing users to customize a large number of configuration options while retaining a core set of functionality. The complexity of the configuration space and the sophisticated interactions among configuration options could easily cause performance issues. A typical configurable system may have hundreds of options, and this generates numerous possible configurations. For example, Apache has more than 1,000 possible configuration options [32]. Unfortunately, developers often do not know how configuration options and their interactions influence the performance of a software system [63].

Prior work has examined the prevalence of configuration issues that have led to performance problems. Han et al. [63] found that more than half of the performance problems (59%) are due to configuration issues. Figure 5.1 shows a real-world, configuration-related performance bug in Apache. When a user configured Apache with a large value for the StartServers option (e.g., StartServers = 60), restarting the server took more time than usual. The root cause of this bug is an unnecessarily expensive loop computation (line 2). To fix this problem, the loop should first check the status of children server processes (line 3 - 6). There is no need to wake these servers (line 7) if they have already exited.

While there has been a lot of research on addressing software performance issues, such as performance profiling [72, 131], testing [18, 165], and debugging [62, 149, 168], it assumes default configurations while ignoring the influence of other configurations. To find and understand configuration-dependent performance problems, developers can benefit from a performance model that summarizes the influence of configuration options on performance.
Given such a model, a developer can understand how configuration options and their interactions influence performance.

Existing techniques on performance modeling for configurable software systems [59, 126, 127] have focused on sampling the configuration space to build performance models based on the sampled configurations and their corresponding performance measures. The accuracy of a model depends on the sampling strategy, performance measures, and learning algorithms. However, these techniques consider the program as a black-box, i.e., they consider only configuration values and performance measures to learn models but ignore the implementation of the program. As a result, these techniques may not accurately identify performance-influencing configuration options without the knowledge of code. In addition, they may not help developers to pinpoint the code locations that cause performance issues.

This paper presents CoProf, a white-box performance profiling approach for configurable systems that analyzes the behavior of configuration options, detects performance-influencing configuration options, and pinpoints inefficient, configuration-dependent code. CoProf consists of two major phases. In the first phase, the approach identifies individual code locations for which performance depends on configuration options. To do that, we gather execution profiles with different values of a configuration option and infer a complexity model that uses option values to predict the execution cost of a performance-sensitive code location, in particular, loops [72] and system calls [17]. The intuition is that code locations such as loops and system calls are more likely to cause performance problems. In the second phase, CoProf summarizes the performance impact of each configuration option across all performance-sensitive code locations and reports a ranked list of performance-influencing options. The result can help developers to understand which configuration options have the most performance impact on the system.

We envision CoProf to be used in at least three cases. First, a developer who is not aware of which configuration options have a high-performance impact can use CoProf to rank the configuration options in terms of their performance impact. In the example of Figure 5.1, StartServers is ranked at the top among all configuration options. Second, a developer can use CoProf to pinpoint code locations whose performance depends on the option value. In Figure 5.1, the offending code location that is relevant to StartServers is first pinpointed in the loop block (Line 2 – 9). The performance bug is further pinpointed by linking both poll() and select() system calls to the dummy_connection() function call inside the for loop, which leads to what the bug report describes as “polling is taking very long time”. Third, a developer or a researcher who wants to build performance models for the whole system can use existing performance modeling techniques [126] but sample only the performance-influencing configuration options identified by CoProf.
CoProf differs from prior work [59, 78, 95, 126] using performance modeling for configurable software systems because it considers the implementation of the program to accurately identify options that are likely to be performance-influencing. In addition, CoProf guides developers toward understanding configuration-specific performance bottlenecks by identifying performance-influencing code locations. CoProf also differs from prior work [18, 72, 131, 141, 149] on performance profiling and performance bug detection by considering the configuration space of a system, instead of assuming a default configuration. Similar to existing profiling techniques, CoProf is based on dynamic analysis and therefore limited to observing the executions triggered by a given set of inputs. The problem of finding suitable inputs for performance analysis [27, 39, 142] is orthogonal to the problem addressed here.

An alternative way to identify configuration options that are relevant to specific code locations is to use static analysis [82]. However, existing static analysis techniques may not be suitable to identify performance-influencing configuration options. For instance, static analysis hardly scales on modern software systems, such as Web Server and Database, due to their large sizes. In addition, these systems are often heterogeneous — they are written in different programming languages and available in different formats (e.g., source code, binary code), but static analysis techniques are usually defined for single-language, self-contained systems. In contrast, CoProf is a dynamic approach in which the identification of performance-influencing configuration options is from program execution profiles and thus can scale to large and heterogeneous software systems.

To evaluate the effectiveness of CoProf, we apply the approach to four popular real-world C/C++ programs. Our results show that CoProf effectively identifies performance-influencing configuration options. We demonstrate that 5.9% of all configuration options have an influence on performance. Compared to a recent approach SPLConqueror [126], CoProf outperforms in 11 out of 13 cases in ranking performance-influencing options. Among the top-5 performance-
influencing configuration options identified by CoProf in each subject, at least one option has been reported as the root cause of performance bugs by developers.

In summary, this paper contributes to the following:

- An automated white-box and dynamic performance analysis approach that can identify the influence of configuration options on performance for highly-configurable software systems.
- A technique that can identify specific code locations that have high-performance influence due to specific system configurations.

In the next section, we introduce the technical background and the problem statement using a motivating example. We then present the detailed algorithms of CoProf in Section 5.3. Our empirical study and results are presented in Sections 5.5 and 5.6, followed by a discussion of our observations in Section 5.7. We present the related work in Section 5.8, and then give our conclusions in Section 5.9.

5.2 Background

This section provides definition and background on configurable systems and performance bugs.

5.2.1 Definitions

A configurable software system $S$ consists of a set $O$ of configuration options, which includes numeric and non-numeric options. For a numeric option $o_n$, its value $v(o_n)$ is within a specific range. For a non-numeric option $o_b$, its value $v(o_b)$ is from a fixed set of values, e.g., True or False.

A usage scenario of $S$ corresponds to a functional unit. CoProf aims to cover different functional units of the program through multiple usage scenarios. For example, typical usage scenarios of Parallel BZIP2 are compressing data and decompressing data. Each usage scenario is associated with a set of configuration option $O'$, where $O' \subseteq O$. The options used in one scenario may not be applicable to another scenario. For example, Parallel BZIP2 uses the -z option to compress data, whereas the –d option is used for decompressing data.

Since performance bugs often require specific input workload to manifest, we identify the workload for each usage scenario separately. For example, in the usage scenario of an HTTP request in the Apache server, one type of workload is the number of issued requests.
5.2.2 Configuration-Related Performance Bugs

Previous work has studied the challenges that configurability creates for handling performance bugs [63]. They report that more than half of 193 studied performance bugs (59%) are due to configuration problems. There are many cases where a misconfiguration can cause poor performance. The root cause can be classified into two types. The first type is about software bugs, which are programming errors [92, 93] in the code. Figure 5.1 is an example of a software bug. The second type of configuration bugs is system-specific, in which the configuration options may not touch source code but are related to hardware, system topology, and the choice of system core libraries. For example, in Apache bug #45834, a misconfiguration of the firewall cuts authentication communications, which freezes the system. However, the previous study [63] has shown that system-specific configurations account for only a small portion (8% to 17%) of all problems. Therefore, in this paper, we focus on implementation bugs, i.e., the first type.

5.3 Approach

This section presents CoProf, a performance profiling approach that helps developers understand how configuration options influence the performance of a system. Figure 5.2 gives an overview of the approach. The input to the approach is a configurable program and a usage scenario that exercises the program. Our approach consists of two phases. In the first phase, CoProf analyzes how the performance of individual code locations (e.g., loops and system calls) depends on configuration options. For simplicity purposes, we only illustrate the approach with loops. To this end, the approach gathers execution profiles for different configurations and infers code location-level complexity models (location-level models for short). A location-level complexity model describes the execution time spent at the code location (e.g., time inside a loop body or time taken by a system call) under different values of a single or interacting configuration options. For example, the model $m_{\text{loop},o} = k \times v(o)$ describes that the execution time in a loop is linearly dependent on the value of a configuration option $o$, where $k$ is a constant.
In the second phase, CoProf summarizes the location-level complexity models for each configuration option obtained from the first phase and computes its performance scores. The output of the second phase contains both a ranked list of individual options and a ranked list of interacting options. A higher rank means that the option or the interaction between options has a stronger influence on the overall performance.

One important property of the first phase of CoProf is that performance profiles are generated on demand. Specifically, CoProf computes the error of a currently inferred location-level model and guides the exploration of new configuration values toward profiles that may improve the model’s accuracy. This iterative, feedback-driven approach is the key to efficiently obtain accurate models instead of blindly sampling the value space of possible configurations.

An important property of the second phase of CoProf is that the performance impact of an option is summarized from the location-level complexity models, as opposed to the system-level complexity model using a black-box approach (e.g., computing the performance measurements...
across the entire program execution [126]). The insight is that low-level code locations can be linked to specific options for facilitating performance bug diagnosis. The complexity orders on individual code locations can more precisely reflect the performance impact of an option than a black-box approach. To illustrate the idea, consider the example below:

<table>
<thead>
<tr>
<th>$o_1$</th>
<th>count($L_1$)</th>
<th>count($L_2$)</th>
<th>count($L_1, L_2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10000</td>
<td>100001</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>10000</td>
<td>100020</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>10000</td>
<td>100030</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>10000</td>
<td>100040</td>
</tr>
</tbody>
</table>

Column 1 indicates the values of option $o_1$, column 2 and column 3 indicate the location-level performance measures for code locations $L_1$ and $L_2$, and the last column indicates the system-level performance measures using a black-box approach. Suppose that the current sample data for $o_1$ is 1, 2, 3, and 4. CoProf can find that the complexity model of $L_1$ is $10^*o_1$ and the complexity model of $L_2$ is a constant $100000$. It would conclude that $o_1$ has performance impact on the system because it has a positive linear relationship with $L_1$. As the value of $o_1$ increases, it will affect the system-level performance. In contrast, under the current sample data, $o_1$ has an almost constant relationship with the system-level performance measures and a black box approach would consider it to have no performance impact on the system.

In the rest of this paper, we use the example in the lower part of Figure 5.2 to illustrate CoProf. This example contains five configuration options: $o_1$ – $o_5$. Option $o_3$ is a binary option while the others are numeric options. The default value of each option is 0.

5.3.1 Inferring Location-Level Complexity Models

In the first phase, CoProf analyzes how the performance of individual code locations depends on the configuration options. We focus on loops and system calls as code locations in this work, because they often influence a system’s performance in significant ways and they are at the core of various performance problems [72, 124].

5.3.1.1 Performance Measurements

Given a usage scenario and a set of configuration options, CoProf measures the performance of each loop and system call in the program. CoProf uses the wallclock execution time as the performance measurement. Recent work on performance modeling [126] and
performance bug detection \cite{155} techniques also used wall-clock execution time as performance measurements. While other performance measurements, such as the number of executed conditional instructions (i.e., direct/indirect calls and direct/indirect branches) or the number of instruction counts, can also be used \cite{108,123}, they may not be representative of the actual performance.

5.3.1.2 Location-Level Complexity Models

Based on performance measurements obtained for different configurations, CoProf infers a complexity model for each loop and system call.

**Definition 1 (Complexity model).** A complexity model \( m \) is a function that predicts the execution time of a code location \( l \) using the value of one or more configuration options. We consider three kinds of complexity models:

- A constant model \( m_l = n \) expresses that the performance cost of the measured code location \( l \) is \( n \), independently of any options.
- A single-option model \( m_l = f(o) \) expresses that the performance cost of the measured code location \( l \) is a function of the option \( o \).
- A model of interacting options \( m_l | o' = 1 = f(o) \) expresses that the performance cost of the measured code location \( l \) is a function of the option \( o \) given that the binary option \( o' \) is enabled.

To infer the function \( f \) of such a model, CoProf analyzes a sequence of measurements. Specifically, the inference takes a sequence of performance measurements \( p_1, .., p_k \), where each measurement is gathered with a different configuration value \( v_1, .., v_k \) for the option \( o \). We explain below how CoProf obtains this sequence of measurements. Given the sequence, the approach uses regression models to fit the performance measurements to the configuration values. CoProf uses the sequential minimal optimization algorithm implemented for support vector machines to learn the complexity model \cite{104}. CoProf fits the data points \((v_i, p_i)\) to linear and nonlinear regression models. To measure how well a model fits data points, the approach computes the mean absolute error and selects the complexity model with the lowest error.

5.3.1.3 Feedback-Driven Profiling and Model Inference

To infer complexity models, CoProf requires performance measurements from different values of configuration options. One possible approach is to first measure performance for a sufficiently large set of configurations and then infer models. However, the downside of this
approach is that performance measurements are taken without knowing what and how many
data points are sufficient for inferring an accurate model. Given the high cost of performance
measuring, which requires executing the program with the given usage scenario, this approach
suffers from a scalability problem.

Instead of the first-measure-then-infer approach, CoProf uses a feedback-driven profiling
and model inference approach. Specifically, the approach incrementally obtains new
measurements to expand the existing profile data to iteratively improve the inferred model. The
main benefit is that the feedback-driven approach requires fewer performance measurements
than the alternative approach, which significantly reduces the overall cost of the model
inference.

Algorithm 1 Iterative inference of location-level complexity models.

Require: Set \( L \) of code locations, set \( O \) of options
Ensure: Map \( U \) of code locations to models
1: \( \text{err} = \text{err}_{\text{prev}} = \infty \)
2: for each \( o_i \in O \) do
3: for each \( l \in L \) do
4: while time < LIMIT do
5: \( P \leftarrow \text{testMeasure}(l, o_i) \)
6: \( M \leftarrow \text{LearnModel}(P) \)
7: \( \text{err} = \text{ComputeError}(M, P) \)
8: if \( \text{err} < \text{thresh}_{\text{err}} \) or \( \text{err} - \text{err}_{\text{prev}} < \text{thresh}_{\text{err}} \) then
9: \( U \leftarrow U \cup \{l \to M\} \)
10: break
11: end if
12: \( P' \leftarrow \text{GenerateNewProfile}(U, o_i) \)
13: \( \text{err}_{\text{prev}} = \text{err} \)
14: \( P \leftarrow P \cup P' \)
15: end while
16: end for
17: end for

Algorithm 1 summarizes the main steps of our iterative model inference approach. The
algorithm takes a set \( L \) of code locations (i.e., loops and system calls) and a set \( O \) of
configuration options as input, and outputs an inferred model for each code location on each
configuration option. Our complexity model inference algorithm starts by obtaining an initial
sequence \( P \) of performance measures for configuration option values \( v_1, v_2, \ldots, v_k \) of each
option \( o_i \) (line 2). We first describe the approach for single options and then generalize it to
interactions of options. CoProf executes each \( o_i \)'s value on the program while keeping the other
options default to assess the performance influence of \( o_i \) only. Therefore, after the execution,
we obtain \( k \) profiles, and have a sequence of measures \( P = p_1, \ldots, p_k \) for each code location.
The main part of the algorithm is the iteration cycle of inferring and refining complexity models
(line 3-16). CoProf first obtains the measures for the code location \( l \) across all execution profiles
(line 5). It then infers complexity models (line 6) based on the option values and performance
measures using the regression models. In the next step, CoProf computes the errors \( \text{err} \) for the
inferred model (line 7). Based on the computed error, the algorithm terminates and returns the
model when either of two conditions is met: 1) The average error is less than a threshold of the
mean absolute prediction error; 2) The improvement of the average error is less than a
threshold of model improvement $threshold_{inc}$ (line 8). If neither of the above conditions is met, CoProf continues to refine the model by generating new profiles given a new set of values for $oi$ (line 12) until a time limit is reached.

In the presence of nested loops, the algorithm selects and profiles both the outer loop and the inner loops in one iteration, which improves the efficiency of model inference. This strategy provides details in individual loops and also avoids misrepresentations in cases when the whole program runs inside a single loop.

**Selecting configuration values.** While the number of sampled option values should be large enough to infer an accurate model, a very large number would require a much longer time to refine the model. Therefore, the initial set of values for $oi$ is generated by following a fixed increment percentage $N\%$ and additional values are generated by adjusting the value of $N$. Specifically, for each numeric option with a value range in $[min, max]$, if $max > 10$, CoProf begins with $min$, and iteratively selects the next value by an increment of $max \times N\%$ until $max$ is reached. Each value is rounded up to an integer. If the error does not reach the threshold, CoProf automatically decreases the increment percentage from $N\%$ to $N/2\%$ to generate more option values. The intuition is that as more data points are used, it can often lead to a lower error in the inferred models. If the option has a smaller value range (e.g., $max \leq 10$), CoProf selects all integer values within the range. The desired size of the training data depends largely on the specific problem. Upon the completion of the algorithm, CoProf updates the average error $err_{pre}$ (line 13). Both new and old profiles are used for learning in the next iteration (line 14).

**Interactions of options.** Prior work has shown that non-numeric options are often external identifiers that are used to control certain functionalities [112]. Switching a non-numeric option $ob$ to a different value may cause the numeric option $on$ to cover new performance-sensitive code locations. To identify such numeric options, CoProf employs a pair-wise strategy, in which each value of the numeric option $on$ is combined with all possible values (except the default value) of each non-numeric option $ob$ while keeping other options with their default values. If the performance-sensitive code location coverage is different from when $ob$ is at its default value, CoProf determines that $ob$ and $on$ have a potential interaction. In this case, CoProf learns complexity models for the loops whose coverage is controlled by $ob$. If the order of a complexity model is linear or higher, $ob$ and $on$ have a confirmed interaction.

CoProf considers pairwise interactions between numeric and non-numeric options. It is possible that a higher-strength interaction (among more than two configuration options) may cause performance problems. However, exhaustive testing of the combinations of all configuration options is not practical due to limited system resources [33]. A previous study [63] showed that the majority (72% to 73%) of configuration-related performance bugs are related
to only one option, 13% to 15% of the bugs are related to two options, whereas about 12% to 15% of the examined parameter configuration bugs involve more than two options. Therefore, we focus on single options and pairwise interactions. Yet, CoProf has advantages over most existing performance modeling techniques [59, 126, 127] that support only binary options and cannot identify performance-related numeric options which are commonly used in the real-world configurable systems [112].

**Running example.** In the example program of Figure 5.2, CoProf first infers complexity models for $o_1$. Suppose the value range of $o_1$ is [0, 1000] and the fixed percentage $N\%$ is 10% [85] (i.e., one may increase $o_1$ by 100 in each round). So the initial sequence of values for $o_1$ is 0, 100, 200, . . . , 1000. The program is then exercised on 11 configurations: \{0,0,0,0,0\}, \{100,0,0,0,0\}, . . . , \{1000,0,0,0,0\}. As a result, we obtain eleven execution profiles: $P_1, P_2, \ldots, P_{11}$, where each profile records the performance measures of performance-sensitive code location $L_1$, $L_2$, and $L_3$ that are covered by the respective configuration. If the error does not reach the threshold or if there is still room to improve the accuracy, $N\%$ is set to 5% , so that additional configurations \{50,0,0,0,0\}, . . . , \{950,0,0,0,0\} are generated and exercised to produce more profiles. When the error reaches the threshold or a timeout occurs, CoProf stops generating profiles and infers the complexity models for $o_1$ on the three code locations: $ml_{1,o_1}$, $ml_{2,o_1}$, and $ml_{3,o_1}$. As Figure 5.2 shows, they correspond to two constant models and a linear model ($ml_{2,o_1}$). Following the same process, CoProf continues inferring the complexity models for the other numeric options (i.e., $o_2, o_4, o_5$).

Next, CoProf infers the complexity models for interacting options. CoProf first changes the default value of $o_3$ from 0 to 1 and pairs $o_3$ with other options (i.e., \{(o_1, o_3), (o_2, o_3), (o_4, o_3), (o_5, o_3)\}). For example, when pairing $o_3$ with the eleven values of $o_1$, the program is exercised on eleven new configurations: \{0,0,1,0,0\}, \{100,0,1,0,0\}, . . . , \{1000,0,1,0,0\}. CoProf then observes that new performance-sensitive code locations (i.e., Location 4–6) are discovered on each of the four interacting options. CoProf begins to infer complexity models for the newly discovered code locations on all configuration options: $ml_{4,o_1} | o_3=1$, . . . , $ml_{5,o_4} | o_3=1$, . . . , $ml_{6,o_5} | o_3=1$. Among these new models, only $ml_{4,o_1} | o_3=1$ and $ml_{5,o_4} | o_3=1$ are non-constant models. Therefore, \{(o_4, o_3)\} and \{(o_5, o_3)\} have true interactions.

5.3.2 Estimating the Performance Impact of Configuration Options

Upon completion of complexity inference, each single configuration option and interacting configuration option $o$ corresponds to a set of complexity models across all distinct code locations, \{$ml_{1,o_1},ml_{2,o_1},\ldots,ml_{n,o_1}$\}, where 1, . . . , $n$ are indices of code locations.

Next, CoProf ranks the configuration options and their interactions based on their performance influence. CoProf employs a weighting strategy to rank single options and their
interactions. The insight is that a single option or an option interaction associated with more higher-order complexity models is more likely to have a stronger performance impact thus having a higher ranking score. In particular, we assign different weights to code locations with different orders (i.e., constant order, linear order, and higher order). The final performance impact of the option \( o, P_o \), is the sum of weighted costs over all code locations.

\[
P_o \sum_{i=1}^{n} (w_{ic(l_i)})
\]

Here, \( l_i \) refers to a code location, \( c(l_i) \) is the performance cost of \( l_i \), a value of the recorded performance measures, and \( w_i \) is the model weight of the code location \( l_i \).

Running example. In the example program of Figure 5.2, we assign weights 1 (20), 2 (21), and 4 (22) to the constant, (positive) linear, and (positive) nonlinear models, respectively. We use \( o_1 \) and \( o_2 \) as examples. \( o_1 \) corresponds to one linear model (Loop2) and two constant modes (Loop1 and Loop3) and \( o_2 \) corresponds to one quadratic model (Loop3) and two constant models (Loop1 and Loop2). Suppose the performance impact of \( o_1 \) (with \( o_3 \) disabled) across Loop1 – Loop3 is \( (c(l_1), c(l_2), c(l_3)) = (5, 10, 5) \) and that of \( o_2 \) is \( (c(l_1), c(l_2), c(l_3)) = (2, 4, 10) \). The final performance impact for the two options is computed as: \( P_{o_1} = 1*5 + 2*10 + 1*5 = 30 \); \( P_{o_2} = 1*2 + 1*4 + 4*10 = 46 \). Therefore, \( o_2 \) has a higher performance impact than \( o_1 \). The table in Figure 5.2 summarizes the number of complexity models under different orders and the ranking of both single configuration options and their interactions.

5.4 Implementation

CoProf first classifies the configuration options into numeric options and non-numeric options (binary or enum). This step is semi-automated. We develop a parsing tool to extract all configuration options and their descriptions from the documentation, and then manually identify the values and value range of these options. This is a onetime effort. CoProf uses the Intel PIN dynamic binary instrumentation framework [84] to identify loops and systems calls to collect performance measurements. To learn location-level complexity models, CoProf uses Linear Regression, Sequential Minimal Optimization (SMO), and Multilayer Perceptron (MP) from Weka [61].

5.5 Experimental Setup

We apply CoProf to 13 usage scenarios of four popular, highly-configurable C/C++ programs. The evaluation aims to address three research questions:
RQ1: How effective is CoProf at detecting and ranking performance-influencing configuration options and identifying the corresponding code segments?

RQ2: How does CoProf compare to an alternative approach?

RQ3: How effective are the location-level complexity models at predicting performance, and how effective is our feedback-driven iterative approach at refining the models?

5.5.1 Benchmark Programs

We used four highly-configurable and popular open-source software projects: Apache Server, Parallel BZIP2, PostgreSQL, and Lighttpd. These programs cover different application domains: web servers, database engines, and data compression utilities. Each program has various numeric and non-numeric options as listed in Table 5.1. These options were extracted from the respective project documentation. For each project, several common usage scenarios that exercise different functionalities were created. Because different functionalities are controlled by different options, for instance, the web page request and server restart operations each have their own sets of options, it is too ambitious to build one performance model that captures all the essence in different usage scenarios. Therefore, for the evaluation, each usage scenario is treated differently, and CoProf learns a separate model for each usage scenario.

<table>
<thead>
<tr>
<th>App</th>
<th>NLOC</th>
<th>#Ops</th>
<th>#Ob</th>
<th>Workload</th>
<th>Scenario</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>2M</td>
<td>227</td>
<td>51</td>
<td>Multiple Connections</td>
<td>AE-S1</td>
<td>Server restart</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Concurrent Requests</td>
<td>AE-S2</td>
<td>HTTP request</td>
</tr>
<tr>
<td>Deflate</td>
<td>2M</td>
<td>233</td>
<td>52</td>
<td>Concurrent Requests</td>
<td>AED-S1</td>
<td>HTTP compression</td>
</tr>
<tr>
<td>Prefork</td>
<td>2M</td>
<td>224</td>
<td>51</td>
<td>Multiple Connections</td>
<td>AP-S1</td>
<td>Server restart</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Concurrent Requests</td>
<td>AP-S2</td>
<td>HTTP request</td>
</tr>
<tr>
<td>Deflate</td>
<td>2M</td>
<td>230</td>
<td>52</td>
<td>Concurrent Requests</td>
<td>APD-S1</td>
<td>HTTP compression</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>3K</td>
<td>28</td>
<td>11</td>
<td>Number of Raw Files</td>
<td>PZ-S1</td>
<td>Compression</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Number of Archived Files</td>
<td>PZ-S2</td>
<td>Integrity test</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Number of Archived Files</td>
<td>PZ-S3</td>
<td>Decompression</td>
</tr>
<tr>
<td>PSQL</td>
<td>651K</td>
<td>222</td>
<td>38</td>
<td>Concurrent Transactions</td>
<td>PS-S1</td>
<td>Select query</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Concurrent Transactions</td>
<td>PS-S2</td>
<td>Update query</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>96K</td>
<td>175</td>
<td>175</td>
<td>Multiple Connections</td>
<td>LH-S1</td>
<td>Server restart</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Concurrent Requests</td>
<td>LH-S2</td>
<td>HTTP request</td>
</tr>
</tbody>
</table>
5.5.2 RQ1: Ranking Performance-Influencing Configuration Options

To evaluate the effectiveness of ranking, we use the mean average precision (MAP). MAP is a single-figure measure of ranked retrieval results independent of the size of the top list [121]. It is designed for general ranked retrieval problems, where a query can have multiple relevant documents. To compute MAP, it first calculates the average precision (AP) for each individual query \(Q_i\), and then calculates the mean of APs on the set of queries \(Q\):

\[
MAP = \frac{1}{|Q|} \sum_{Q_i \in Q} AP(Q_i)
\]

To illustrate the MAP calculation, suppose that two configuration options \(o_1\) and \(o_2\) are ground truths. If Technique-I ranks the two options at the 1st and 2nd positions among all 500 options and Technique-II ranks the two options at the 1st and 3rd positions, then the MAP of Technique-I is \((1/1 + 2/2)/2 = 1\) and the MAP of Technique-II is \((1/1 + 2/3)/2 = 0.8\).

To determine the ground truth, we manually inspect the documentation and bug reports of the benchmark programs to identify configuration options that have a performance influence and use these options as the ground truth. For example, the Apache documentation states that when setting the option DeflateCompressionLevel, “the higher the value, the better the compression, but the more CPU time is required to achieve this.” [10]. Therefore, we consider DeflateCompressionLevel to be a true performance influencing option. Furthermore, we manually examine bug reports and code patches to verify whether any of the code locations associated with the performance-influencing options have bugs. We consider a code location has bugs if (i) it was reported and confirmed by developers and (ii) the problem had been fixed in the subsequent releases.

5.5.3 RQ2: Comparison with A Baseline Technique

To answer RQ2, ideally, the comparison should be done with existing approaches that detect and/or rank performance-influencing configuration options. However, we cannot find an existing approach with this specific goal. As discussed in Section 5.1, performance modeling techniques sample configuration options to build models to predict a system’s performance. In theory, options in the models indicate their impact on the performance. Therefore, we compare CoProf to the performance modeling approach using just configuration options. To learn performance models for configurable software systems, SPLConqueror [126] is the only available tool to the best of our knowledge. SPLConqueror is a black-box approach that uses
machine learning and heuristic sampling to learn a performance model from a set of configuration option values.

To illustrate SPLConqueror, we consider a server program with options for defining the maximum number of client requests \( r \) and a binary option \( p \) for enabling HTTP persistent connection. A configuration-aware performance model for this system is:

\[
M = 3 \cdot v(r)^2 - 2 \cdot v(p) + 10
\]

In the above example, the two options influence the performance to different degrees. The term \( 3 \cdot v(r)^2 \) indicates that the maximum number of client requests influences the overall running time in a quadratic way. The term \( 2 \cdot v(p) \) denotes that the HTTP persistent connection option, if enabled, speeds up the execution time by two time units, e.g., seconds. Since \( 3 \cdot v(r)^2 \) term has the higher order, \( r \) is ranked higher than \( p \) in terms of performance impact. However, since SPLConqueror is a black-box approach, it cannot identify the code regions linked to the configuration options.

Therefore, we use only the term orders generated from the SPLConqueror performance model to detect and rank performance-influencing configuration options.

5.5.4 RQ3: Effectiveness of Feedback-Driven Learning

To answer RQ3, we measure the accuracy of location-level models. We calculate the mean absolute error rate (MAE):

\[
MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}
\]

which compares the model prediction \((y_i)\) to the actual outcome \((x_i)\).

5.5.5 Study Operation

We set the threshold of the mean absolute error to 0.1 and the threshold of error improvement to 25%. These thresholds are empirically chosen because they yield an accurate model in a reasonable time (within 24 hours). To obtain the execution profiles (Section 5.3.1.3), the percentage used for generating the initial set of values of each numeric option is set to 10%. We assign a linear set of weights: 1, 2, and 4, to the constant, (positive) linear, and (positive) nonlinear models, respectively. To control the variance due to the randomness in each technique, we run each experiment three times and report the average result.
Table 5.2 Performance-Influencing Configuration Options Ranking

<table>
<thead>
<tr>
<th>Scen.</th>
<th>CoProf</th>
<th>SPLConqueror</th>
<th>GT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-5</td>
<td>Top-5</td>
<td></td>
</tr>
<tr>
<td>AE-S1</td>
<td>G.S.T., KeepAliveTimeout, L.I.R., L.R.B., L.R.F.</td>
<td>0.5 M.C., R.L.N.P., M.S.S., R.B.S.</td>
<td>0 [11]</td>
</tr>
<tr>
<td>PZ-S1</td>
<td>FileSize, MaxMem, B.S., NumOfProcessor, L.A.</td>
<td>0.9 FileSize, B.S., NumOfProcessor</td>
<td>0.8 [100]</td>
</tr>
<tr>
<td>PZ-S2</td>
<td>B.W.T., B.S., C.S.S., MaxMem, NumOfProcessor</td>
<td>0.3 MaxMem, F.S., B.S., NumOfProcessor</td>
<td>0.8 [100]</td>
</tr>
<tr>
<td>PZ-S3</td>
<td>NumOfProcessor, B.S., B.W.T, MaxMem, L.A.</td>
<td>0.8 C.S.S., NumOfProcessor, B.S., MaxMem</td>
<td>0.5 [100]</td>
</tr>
<tr>
<td>PS-S1</td>
<td>A.W.M., T.F.L., C.D, M.C., M.W.M.</td>
<td>0.5 S.S.L., M.P.T., S.P.S., A.V., L.D.</td>
<td>0 [102] [103]</td>
</tr>
<tr>
<td>PS-S2</td>
<td>C.D., M.F.P.P., A.W.M., M.C., T.F.L.</td>
<td>0.3 M.P.W.P.G., S.S.L.</td>
<td>0 [102] [103]</td>
</tr>
<tr>
<td>LH-S1</td>
<td>C.K., S.K., S.MC, S.L.B., S.MD</td>
<td>0.2 D.I.R.H, C.K., S.M.R., S.L.B.</td>
<td>0 [80]</td>
</tr>
<tr>
<td>LH-S2</td>
<td>S.MRS, S.MKAR, S.LB, S.MC, S.MD</td>
<td>0.5 S.MRI, S.LB</td>
<td>0 [80]</td>
</tr>
</tbody>
</table>

5.6 Results

5.6.1 RQ1: Ranking Performance-Influencing Configuration Options

CoProf ranks configuration options in terms of their performance impact. The ranked list of options can be used by developers to understand and debug performance issues. Note that a performance-influencing option does not always imply a performance bug. It only indicates that the option may generate a high-performance impact on the system.

Based on the ground truth, we assessed the effectiveness of CoProf’s ranking configuration options. Column “Top-5” of Table 5.2 shows the top-5 configuration options ranked by CoProf. The options marked in bold font indicate they are known to be performance-influencing. The options that are marked with “⋆” correspond to known performance bugs. The “GT” column shows the sources of the ground truth. Overall, the results show that the top-5 options of each
usage scenario include at least one known performance-influencing options and CoProf is effective in ranking performance-influencing configuration options.

**Examples AP-S1.** In Apache bug #54852 and #34508, users reported that a higher value of configuration option “StartServers” would cause a slow down during a graceful server restart. CoProf ranked StartServers at the top among all 224 configuration options. This usage scenario was associated with 50 constant models, eight linear models, and five higher-order models. The loop implementation that had caused the bug (Figure 5.1) was among one of the linear location-level models. Also, the function “dummy_connection()” called inside the loop utilized system calls poll() and select() internally. The system call select() took on average 500 milliseconds to complete whereas most system calls use less than one millisecond, therefore configuration option “StartServers” had the highest performance impact.

**AW-S2.** In Apache bug #42031, when the number of HTTP requests reached the MaxClients limit, Apache child processes would start to freeze. This bug was related to the options KeepAliveTimeout and MaxClients, which were ranked by CoProf at position four and five, respectively. The bug had been fixed by adding locks to a while loop which had a linear relationship with the option KeepAliveTimeout.

### 5.6.2 RQ2: Comparison with a Baseline Technique

Column “Top-5” under “SPLConqueror” of Table 5.2 shows the top-5 performance-influencing options in the performance model built by SPLConqueror. Column “MAP” under CoProf and SPLConqueror shows the MAP scores. Compared to SPLConqueror, the MAP score in CoProf is higher in 11 out of 13 scenarios ranging from 0.2 to 0.9 and averaging 0.5. The results show that CoProf can achieve better configuration option ranking than SPLConqueror.

**Table 5.3 Location-Level Complexity Models**

<table>
<thead>
<tr>
<th>Usage Scenario</th>
<th>Error</th>
<th>#Models</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Begin</td>
<td>End</td>
<td>Constant</td>
</tr>
<tr>
<td>AE-S1</td>
<td>13.9</td>
<td>9.1</td>
<td>16</td>
</tr>
<tr>
<td>AE-S2</td>
<td>2.6</td>
<td>1.7</td>
<td>17</td>
</tr>
<tr>
<td>AED-S1</td>
<td>25.5</td>
<td>3.9</td>
<td>19</td>
</tr>
<tr>
<td>AP-S1</td>
<td>68.2</td>
<td>16.4</td>
<td>11</td>
</tr>
<tr>
<td>AP-S2</td>
<td>34.7</td>
<td>6.2</td>
<td>18</td>
</tr>
<tr>
<td>APD-S1</td>
<td>11.4</td>
<td>5.4</td>
<td>30</td>
</tr>
<tr>
<td>PZ-S1</td>
<td>13.8</td>
<td>4.6</td>
<td>6</td>
</tr>
<tr>
<td>PZ-S2</td>
<td>21.8</td>
<td>11.1</td>
<td>5</td>
</tr>
<tr>
<td>PZ-S3</td>
<td>13.4</td>
<td>2.1</td>
<td>4</td>
</tr>
<tr>
<td>PS-S1</td>
<td>32.1</td>
<td>2.5</td>
<td>49</td>
</tr>
<tr>
<td>PS-S2</td>
<td>7.4</td>
<td>1.7</td>
<td>36</td>
</tr>
<tr>
<td>LH-S1</td>
<td>14.5</td>
<td>3.3</td>
<td>11</td>
</tr>
<tr>
<td>LH-S2</td>
<td>10.8</td>
<td>2.3</td>
<td>21</td>
</tr>
</tbody>
</table>
5.6.3 RQ3: Effective of Feedback-Driven Learning

To answer RQ3, we measured prediction errors of the learned complexity models and to what extent the error reduced due to the iterative model refinement. Columns 2-7 of Table 5.3 show the results, where the “Begin” and “End” columns show the error before and after refinement, respectively. When applying the model refinement, the errors of the inferred models were substantially reduced (57%, on average) across location-level models over all subject scenarios. In summary, the results show that our iterative inference algorithm (Algorithm 1) improved the prediction accuracy when learning the location-level complexity models.
Figure 5.3 Code Location Errors on AE-S1 and PS-S1

**Examples.** Figure 5.3 provides a detailed view of the prediction errors over all configuration options across different performance-sensitive code locations. Due to space limitations, we show results for only AE-S1 and PS-S1 on 10 randomly selected locations. The labels on the horizontal axis are loops marked by unique IDs. Each box reflects the errors measured across all configuration options. Each model is associated with two boxes, where the gray box (left) shows the errors under the initial option values and the white box (right) shows the errors after the refined option values. The results show that CoProf’s model refinement reduces prediction errors across different configuration options.

5.7 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our subjects. Other subjects may exhibit different behaviors. We reduce this threat to some extent by studying subjects from different application domains and those have varying numbers of configuration options.

A threat to internal validity for this study is the possible faults in the implementation of our approach and in the tools that we use to perform the evaluation. We controlled this threat by extensively testing our tools and verifying their results against a smaller program for which we can manually determine the correct results.

A threat to internal validity for this study is the possible faults in the implementation of our approach and in the tools that we use to perform the evaluation. We controlled this threat by extensively testing our tools and verifying their results against a smaller program for which we can manually determine the correct results.

One threat involves the human factor for determining whether a configuration option is performance-influencing. To reduce this threat, we first searched the application documentation and issue trackers for performance-influencing options and then compared these options to the top-5 options reported by CoProf.

A second threat is the relationship between performance impact and option values in a model. CoProf considers constant, linear, and higher-order relations. It is possible that the performance impact and the option values can have an inverse relationship. We did not observe such relationships in our study. We plan to study the cost-effectiveness of using inverse models in future work.

For the third threat, CoProf focuses on performance-sensitive code locations because performance bugs due to loops and system calls are more pervasive [72]. Other code locations
may also cause performance problems, such as multi-threaded code that causes lock contention [138, 162, 168] and (indirect) recursion. As part of the future work, we plan to incorporate these code locations into CoProf.

5.8 Related Work

Numerous research efforts have been made to analyze and detect performance bugs [72, 73, 93, 96, 107, 145]. Jin et al. [72] apply a rule-based pattern matching to detect performance anomalies under normal executions. Jovic et al. [73] propose Lag Hunting, a technique used to collect performance information of deployed Java GUI applications in interactive each session. Such sessions are combined and analyzed to generate reports about potential performance problems automatically. Nistor et al. [93] design an automated performance bug oracle Toddler using repetitive memory-access on loop iterations. Olivo et al. [96] propose a static analysis to identify redundant travel performance bugs. Pradel et al. [107] present SpeedGun, an automatic performance regression testing technique. Unlike our technique, prior work does not provide insights on how configuration options affect performance through performance-sensitive code locations.

Several test case generation techniques have been proposed to generate large workload test inputs [27, 54, 164]. Burnim et al. [27] present an automatic performance test generation technique that attempts to construct an input that triggers the worst-case computational complexity. Grechanik et al. [54] learn rules from execution traces of applications and then use these rules to automatically select test input data to find performance problems. Zhang et al. [164] apply symbolic execution to build program performance models to generate load tests. Unlike our work, these techniques assume default configurations while ignoring the influence of other configurations.

There has been a great body of work on constructing performance models for various purposes [67, 78, 139, 155]. Huang et al. [67] propose an approach to build prediction models for program performance using profile data generated from sampled input files and command-line arguments. Kwon et al. [78] present a framework to predict the performance of Android applications by constructing automatically generated executable code snippets. Tarvo et al. [139] propose a method to build performance models for multithreaded programs. Xiao et al. [155] design the DeltaInference framework which is primarily used for inferring workload-sensitive loop counts in graphical user interface applications. Unlike our work, however, these prior work do not target using configuration options to build performance models.

There has been some research on learning-based approaches that analyze the performance of configurable software systems [59, 95, 126]. Guo et al. [59] propose an approach to predict a
configuration’s performance based on random sampling and a statistical learning method CART to build performance models that incorporate feature interactions.

Oh et al. [95] develop a random sampling approach to find configurations for achieving the optimal performance in software product line models. Such methods only consider the presence of specific configuration options while ignoring the influence of their values on system performance.

Siegmund et al. [126] propose performance-influence models that can associate system-level performance with the influence of individual configurations and their interactions. While these techniques provide good insights about factors involved in analyzing configuration-related performance problems, they are incapable of pinpointing their associated implementations that can influence the performance of systems. The technique assumes that the performance-influencing options are readily available before building the model. In that respect, our method is orthogonal to the prior approach and complimentary.

Velez et al. [145] propose ConfigCrusher to analyze on how configuration options may influence the performance of a software system using static data-flow analysis. Unlike the prior work, our approach uses dynamic analysis techniques, which can scale on large and complex real-world configurable software systems, as shown in the experiment. In addition, our approach can help developers pinpoint the code locations that may cause performance issues.

5.9 Conclusion

We present CoProf, a configuration-aware profiling approach that helps developers to understand and identify performance-influencing configuration options. The approach summarizes the performance behavior from multiple code location executions to output a ranked list of configuration options. Our study shows that CoProf can successfully identify performance-influencing options in 11 out of 13 cases with four real-world highly-configurable software systems when compared to a recent blackbox performance modeling technique. We envision that our work serves as a basis for future configuration-aware performance analysis techniques, in particular, for the currently understudied challenge of debugging configuration-specific performance problems.
Software performance is important for ensuring the quality of software products. Performance bugs, defined as programming errors that cause significant performance degradation, can lead to slow systems and poor user experience. While there has been some research on automated performance testing such as test case generation, the main idea is to select workload values to increase the program execution times. These techniques often assume the initial test cases have the right combination of input parameters and focus on evolving values of certain input parameters. However, such an assumption may not hold for highly-configurable real-word applications, in which the combinations of input parameters can be very large. In this chapter, we manually analyze 300 bug reports from three large open source projects – Apache HTTP Server, MySQL, and Mozilla Firefox. We found that 1) exposing performance bugs often requires combinations of multiple input parameters, and 2) certain input parameters are frequently involved in exposing performance bugs. Guided by these findings, we designed and evaluated an automated approach, PerfLearner, to extract execution commands and input parameters from descriptions of performance bug reports and use them to generate test frames for guiding actual performance test case generation.

6.1 Introduction

Software performance is critical to the quality of a deployed system. A performance bug can cause significant performance degradation [17], leading to problems such as poor user experience, long response time, and low system throughput [26, 62, 72, 92, 149]. Compared to functional bugs that typically cause system crashes or incorrect results, performance bugs are substantially more difficult to handle [17, 38] because they often manifest themselves by special inputs and in specific execution environments [92, 96]. Over the past decade, numerous research efforts have been made to analyze, detect, and fix performance bugs [27, 62, 72, 73, 93, 96]. For example, many profiling techniques [72] have been proposed to dynamically determine what program entities (e.g., methods) are responsible for the excessive execution time and resource consumption given an input.

Profiling methods depend on the chosen set of input values, which is a known weakness [125] for successfully detecting performance bugs in the subject under test. To address this problem, several test case generation techniques have been proposed to generate large workload test inputs for increasing the chance of exposing performance bugs [27, 107]. However, there are several limitations in existing performance test generation techniques – many techniques focus on evolving the values of certain input parameters while keeping the other parameters as default. For example, Burnim et al. [27] focus on increasing the workload values of data inputs while keeping the values of configuration options as default. These...
techniques may be ineffective at detecting performance bugs due to combinatorial effects of different input parameters. For example, in Apache bug #52914, the performance bug is exposed only when the configuration options KeepAlive and RequestReadTimeout are specified. Otherwise, by using the default configuration, this performance bug cannot be triggered even if the workload (e.g., the number of requests) is increased.

While a full performance testing with all combinations of input parameters can address the above problem, it is infeasible due to the enormous combination space. For example, the latest version of Apache HTTP Server has 618 input parameters (610 configuration options and 8 types of data inputs). It is impractical to try all combinations of values for these input parameters. To reduce the cost of performance testing, Shen et al. [125] use a genetic algorithm (GA) as a search heuristic for obtaining combinations of input parameter values that maximize the execution time. However, this technique evolves all input parameters, which can be inefficient because many parameters may not provide contributions to the application’s performance.

The goal of our research is twofold. First, we want to understand to what extent performance bugs are related to the combinations of input parameters. A study on performance bug reports from bug tracking systems, such as Bugzilla, can help us understand the characteristics of input parameters and their contributions to performance bugs. Second, we aim to develop a framework to automatically generate combinations of input parameters, also called test frames (discussed in Section 6.2), for guiding the generation of actual performance test cases. To the best of our knowledge, no existing research achieves the same goal.

Our main idea is to mine information from the application’s bug reports to identify commands (i.e., commands for executing the program) and input parameters (i.e., configuration options and data inputs) that have caused performance bugs and use them to generate test frames for testing newer versions of the application. PerfLearner is used during software maintenance and evolution, where the projects’ issue tracking systems have been established. Specifically, we extract and rank commands and input parameters from each bug report. We then generate test frames (a combination of the commands and input parameters) for each bug report and prioritize the most frequently generated test frames among all bug reports. Our hypothesis includes: 1) bug reports contain a specific set of vocabulary related to commands and input parameters that can make the automated text extraction possible; 2) commands and input parameters appearing frequently in performance bug reports may be more likely to trigger performance bugs than the infrequent ones. PerfLearner is applicable software projects with established issue tracking systems.

In this research, we manually identified and analyzed 300 performance bug reports from three popular open source projects. We discovered that it is possible to leverage information retrieval and natural language processing techniques to extract commands and input
parameters from bug reports. We found that some input parameters are more likely to cause performance bugs and should be used with higher priority in performance testing. Based on our findings, we develop PerfLearner, an approach that combines natural language processing and information retrieval to automatically extract relevant commands and input parameters from bug reports and use them to generate performance test frames for guiding performance testing.

In summary, our paper makes the following contributions:

- We develop a tool, PerfLearner, that can automatically extract performance-related commands and input parameters, and generate performance test frames from the bug reports. To the best of our knowledge, this is the first work that automatically generates test frames from bug reports written in natural language.
- We implement PerfLearner and conduct an empirical study to demonstrate its effectiveness and efficiency in generating performance test frames and detecting real performance bugs.

We envision the approach to be applied to at least two scenarios. First, given a performance bug report, a developer who wants to know the commands and input parameters that have caused this bug, may analyze the bug report with PerfLearner. Second, a testing engineer can use PerfLearner to generate and prioritize performance test frames from the historical performance bug reports. The test frames can be converted into actual test cases by giving input parameters with concrete values. Note that PerfLearner is orthogonal to existing performance testing tools. Existing tools focus on increasing the values of certain workload-sensitive input parameters while assuming the test frames (i.e., the combination of input parameters) exist. Therefore, PerfLearner can be used to enhance the effectiveness and efficiency of existing performance testing tools.

To evaluate the approach, we apply PerfLearner to 300 bug reports collected from Apache HTTP Server, MySQL, and Firefox bug tracking systems. Our results show that PerfLearner is able to extract commands and input parameters from performance bug reports with a high accuracy. When using PerfLearner to generate test frames, compared to a state-of-the-art combinatorial testing (CT) technique, it generates significantly less (59.5%) test frames on average to get the ground truth test frame. When combining PerfLearner with an existing performance test input generation tool [125] to test 10 randomly selected performance bugs, PerfLearner detects seven out of 10 bugs within a reasonable time whereas when using the test input generation tool alone failed to detect all 10 bugs.
6.2 Background

The concept of test frame was first introduced in the category-partition method with test specification language (TSL) [97]. TSL was created to define combinations of program input parameters and environment factors. Each combination is a test frame that can be converted into actual test cases. A performance test frame consists of three input categories: command, configuration, and data input. A test frame can have one command in the command category, zero or more configuration options in the configuration category, and zero or more data inputs in the data input category. Each command, configuration option, and data input in a test frame is generally referred to as a test frame element or frame element.

We define a command as an action to execute a functional unit [97] of the program. For example, the MySQL server has several data manipulation commands, including SELECT, UPDATE, and INSERT. These commands correspond to three different functional units: retrieve, modify, and add data records. We define input parameters as explicit input points along with the command. An input parameter can be a configuration option or a data input. Configuration options refer to a set of predefined options, e.g., command-line options or directives in a configuration file. Data inputs refer to the user-supplied data that is processed by the command. For example, the data input associated with the command UPDATE is the name of a table COLUMN.

Figure 6.1 shows a performance bug report snippet with the associated test frame and a test case. The test frame for manifesting this performance bug involves three frame elements: a command UPDATE, a configuration option innodb_fill_factor, and a data input COLUMN. A frame element can be workload-sensitive. In this example, the UPDATE command is workload-sensitive because a large number of UPDATE queries is required to trigger the performance bug. In MySQL, a workload can be simulated by benchmark tools such as mysqlslap. Since many performance test generation techniques have been focusing on identifying the workload-sensitive inputs [54, 155], pinpointing the workload from a bug report may speed up this process for performance test case generation techniques. The actual test case is created by assigning concrete values to frame elements.
6.3 Characteristics

Before designing our approach, we wish to understand to what extent performance bugs are related to certain commands and input parameters.

6.3.1 Data Collection

We chose three large open source software projects: Apache HTTP Server, MySQL Database Server, and Mozilla Firefox Browser. With publicly accessible source code and well-maintained bug tracking systems, these projects have been widely used as subject programs by existing bug characteristic studies [72, 160, 163].

We collected performance bugs from bug tracking systems of Apache, MySQL, and Firefox. We searched these 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” [63]. 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, we selected reports with the bug status field marked as either “RESOLVED”, “VERIFIED”, or “CLOSED” and the resolution field marked as “FIXED”.

The whole process yielded a total of 1383 bugs. With a large amount of the returned bug reports, we calculate the needed sample size is 300, given a confidence level of 95% and a confidence interval of 5. This sampling strategy has been commonly used by existing work [13, 56].

We manually examined 300 bugs in a random order, and during the manual inspection, we follow those reports that have sufficient bug description details and discussions posted by commentators. For each bug report, we try to identify commands, configuration options, data inputs, and workload that cause the performance bug.

To ensure the correctness of our results, the manual inspection was performed independently by two inspectors – graduate students who have two to four years of industrial web development experience with Apache, MySQL, and Firefox. We hold two training sessions of 30 minutes each to explain to inspectors the test frame elements to be extracted from the bug report. Each inspector is given the same set of bugs each week to write down what they consider to be the command, configuration options, data inputs, and workload that trigger the bug in the report. Inspectors met 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.
We refer to the consensus outcome as ground truth frame elements for the bug reports. This process terminates for each subject after 100 bug reports have been included in the sample dataset.

The number of bugs sampled is similar to recent works on performance bug study [38, 63, 72, 163]. While a larger number of bug reports may yield a better evaluation, the cost of the manual process is high — our data collection process took a total of 320 to 400 hours spanning across more than 10 weeks. Columns 1-3 of Table 6.1 list the subject programs, the number of bugs returned by the keyword search, and the number of performance bugs sampled after manual inspection. Columns 4-7 list the number of commands, configuration options, and data inputs available in all three subjects. The full lists of the three categories are saved in separate frame element databases, including command database, configuration database, and data input database. We collected such information by studying all artifacts that are publicly available to users, including documents (e.g., user manuals and online help pages), configuration files, and source code. Each database can be updated separately to accommodate changes in different application versions.

Table 6.1 Workaround Efficiency and Effectiveness

<table>
<thead>
<tr>
<th>Application</th>
<th>Searched Bugs</th>
<th>Sampled Bugs</th>
<th># of CMD</th>
<th># of CO</th>
<th># of DI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>428</td>
<td>100</td>
<td>10</td>
<td>610</td>
<td>8</td>
</tr>
<tr>
<td>MySQL</td>
<td>455</td>
<td>100</td>
<td>11</td>
<td>1240</td>
<td>5</td>
</tr>
<tr>
<td>Firefox</td>
<td>500</td>
<td>100</td>
<td>24</td>
<td>563</td>
<td>17</td>
</tr>
<tr>
<td>Total</td>
<td>1383</td>
<td>300</td>
<td>45</td>
<td>2413</td>
<td>30</td>
</tr>
</tbody>
</table>

6.3.2 Results Analysis

After manually analyzing 300 bug reports, we summarize the following findings:

- A majority (89% to 92%) of studied performance bugs involves more than one input parameters (i.e., configuration options and data inputs): 91% in Apache, 92% in MySQL, and 89% in Firefox. These results imply that combinatorial effects among input parameters should be considered in performance testing.
- A significant number (41%) of performance bugs are related to configurations: 58% in Apache, 41% in MySQL, and 25% in Firefox. These results are consistent with a recent performance bug study [63].
- Only 23% of bugs require specific workload values to manifest: 21% in Apache, 29% in MySQL, and 19% in Firefox. These results imply that workload is only part of the
requirement for exposing performance bugs; other factors, such as configuration options, should also be considered for performance testing.

- Among all 45 commands and 30 inputs for the three software projects, 55% of them appear more than three times in the studied bug reports. Among all 2413 configuration options for the three software projects, only 5% of them are related to the studied performance bug reports and 57% of them appear more than one time. These results suggest that a small subset of configuration options tend to affect application’s performance, so performance testing might focus mainly on such options to improve the efficiency of testing. In addition, test frame elements that appear multiple times in performance bug reports might be more likely to cause performance bugs than the others and should be given higher priority in performance testing.

6.4 PerfLearner Approach

Guided by the findings in Section 6.3, we design and develop PerfLearner, an automated approach for extracting performance test frames from bug reports. Figure 6.2 shows an overview of the PerfLearner framework. PerfLearner consists of three steps: frame element extraction, test frame generation, and performance test case generation. The shaded boxes indicate the information supplied by users.

Frame Element Extraction. Given a performance bug report, PerfLearner automatically extracts frame elements and their associated workload from the report. PerfLearner assumes that a bug report has already been labeled as “performance bug”, although existing techniques on classifying bug reports [51, 148, 154] can be adopted to automatically classify performance bugs. The list of frame elements are application and domain-specific, e.g., each application is associated with a list of different configuration options. The bug corpora for each application is built from sources described in Section 6.3.1. The output of this step is a list of ranked frame elements and their associated workload (if any) under each input category for each bug report.

Performance Test Frame Generation. PerfLearner utilizes ranked frame elements, a strength file, and a constraint file to generate performance test frames. The strength file, which is used to restrict the number of test frames, specifies the strength of interaction among elements within each input category. The constraint file specifies the constraints among frame elements to ensure their combinations are valid. Both files are defined once by developers for each application, and generic to all bug reports in the same application. Next, PerfLearner generates a set of test frames for each performance bug report by combining the selected commands and input parameters with respect to the strength and constraint files. These test frames are closely related to the performance bug described in the report. Finally, PerfLearner
counts the frequency of test frames generated from all bug reports and ranks them in a descending order. The top-ranked test frames are used first to generate performance test cases.

Performance Test Case Generation. PerfLearner iteratively selects a test frame from the ranked test frames and converts it into actual performance test cases by assigning frame elements with concrete values. PerfLearner can be combined with existing performance testing tools, such as profiling and test generation tools. The current version of PerfLearner is combined with a performance test input generation tool [125] that uses a search-based algorithm to automatically generate input values to expose performance bugs.

Figure 6.2 Overview of PerfLearner

6.4.1 Test Frame Element Extraction

For each bug report that is labeled as a performance bug report, PerfLearner extracts commands, configuration options, data inputs, and the associated workload. A straightforward approach is to match frame element databases against each bug report using a “grep”-like method. The matched elements can then be ranked by counting their occurrences – the element with the highest count is more likely to be the ground truth frame element for the performance bug. However, in a bug report written in natural language, many words can be ambiguous in their meaning – the same word can refer to a command or a configuration option depending on the context. For example, in the Apache bug #52914, the word token timeout can be matched as either a command or a configuration option. In addition, simply counting the occurrence of a token may result in false positives. In the Apache bug #52914, both start and request appear in the bug report, so both tokens would be matched as commands of this bug report. Incidentally, the count of start is actually higher than the count of request, although the ground truth command is request.

PerfLearner employs two strategies to address the above problems. First, PerfLearner uses natural language processing and information retrieval, together with user manuals to address the mismatch problem between the frame elements (query) and bug reports (documents).
Second, we summarize 18 linguistic patterns that are commonly used to describe commands (eight patterns), input parameters (four patterns), and workload in bug reports (six patterns). While the frame elements are application-specific, the linguistic patterns are generic and hence can be reused for different applications.

To avoid overfitting, the first author summarized the linguistic patterns from the 1083 bug reports (excluding the 300 sampled bug reports in the dataset). In the experiment, these patterns are applied to the 300 bug reports. We can automatically detect the presence of these patterns to locate sentences describing a particular input category and identify the frame element under that category more accurately. Table 6.2 shows the number of patterns we identified in all sentences from the 300 bug reports. While there has been some research on using linguistic patterns in other software activities, such as analyzing developer intention [40, 76] and detecting missing information [30], little work is known on using linguistic patterns to identify commands and input parameters.

<table>
<thead>
<tr>
<th>Application</th>
<th># of Matched Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Commands (8)</td>
</tr>
<tr>
<td>Apache</td>
<td>104</td>
</tr>
<tr>
<td>MySQL</td>
<td>228</td>
</tr>
<tr>
<td>Firefox</td>
<td>203</td>
</tr>
</tbody>
</table>

6.4.1.1 Commands

We observe that a command often appears with the bug symptom in one sentence. For example, the sentence describing the symptom of Apache bug #52914 is “I could reproduce the 100% CPU with POST requests”, where the symptom is “100% CPU” and the command is request. If we identify sentences containing bug symptoms, it narrows down the search and improves the accuracy of finding the performance bug-triggering commands.

We have defined six linguistic patterns using the part-of-speech tag for detecting (one or more) sentences containing symptoms. If such sentences are detected, PerfLearner matches the command against these sentences and counts their occurrences. The identified \( k \) commands are ranked at the top \( k \) position in a descending order with respect to their occurrences.

Our patterns can precisely identify commands in 91% performance bug reports (i.e., ranked at the top-1), compared to the 78% precision rate by the “grep”-like method. The most frequently used pattern, as seen in Figure 6.3, illustrates a pattern that uses a verb and a phrase, where the verb refers to the command element and the phrase refers to the predefined list of phrases indicating performance bug symptoms. If any of the symptoms appear in the sentence,
the verb is identified as a candidate of the bug-triggering command. If no symptom sentences are detected, PerfLearner prioritizes sentences that appeared in the bug report title as well as the first post, and uses the "grep"-like method to count the occurrences of commands. If no command sentences are detected, the same approach is applied to the entire bug corpus.

![Definitions: CMD e{drop, create, ...}, SYMP e{slow, long, ...}. Pattern: CMD + SYMP. Description: Command verbs appear in the same sentence that symptoms exist. Example: [DROP] on very large tables can be very [slow]].

Figure 6.3 A Common Pattern to Identify Command

6.4.1.2 Data Inputs

PerfLearner ranks data inputs in a similar way as commands because simply matching a bug report against the elements in data input is imprecise. PerfLearner defines four linguistic patterns to detect sentences that contain data inputs and rank data inputs within these sentences. Figure 6.4 shows one of the commonly used patterns. This pattern indicates that data inputs coexist with commands in the same sentence. Specifically, the sentence starts with a command (i.e. update), followed by a preposition (i.e. to, on) and the data input (i.e. column).

![Definitions: ADP e{to, on, ...}, NOUN e{DATA INPUT}. Pattern: CMD + ADP + NOUN. Description: Data input is identified as the subject of the command. Example: update [CMD] to [ADP] indexed [NOUN] much slower in 5.7.5.

Figure 6.4 A Common Pattern to Identify Data Inputs

6.4.1.3 Configuration Options

Unlike commands and data inputs, we observe that many configuration options cannot be directly searched from bug reports. One solution is to leverage information retrieval (IR) algorithms such as TF-IDF [105] and cosine similarity [48] based on the vector space model (VSM) to rank configuration options in terms of their relevance to the bug report. A straightforward method is to split the configuration name into tokens to calculate its cosine similarity to the bug report. However, we observe that many configuration options share with the same tokens. Since a configuration option name is often short, this approach may result in many equally ranked configuration options. For example, in Figure 6.1, innodb_buffer_pool_instances and innodb_buffer_pool_size would be ranked equally if “innodb”, “buffer”, and “pool” are the three word tokens appearing in the report. To improve the accuracy of ranking, we leverage manuals that describe configuration options to bridge the lexical gap between configuration option names and bug reports. In the example of Figure 6.1, the manual description of innodb_fill_factor (Figure 6.7) contains words such as “b-tree”,
“index”, and “space”, which also appear in the bug report, can be used to link the configuration option to the bug report effectively.

To compute the similarity between a configuration option $o$ and a bug report $br$, we first concatenate $o$ with its textual description, where $o = o \cup o.desc$. PerfLearner then processes $o$ by standard NLP pre-processing steps: word tokenization and stop word removal. The tokenization converts bug reports into a “bag of words” using white spaces. We then remove punctuation, numbers, and standard stop words. Compound words, such as the configuration option name Browser.chrome.image_icons.max_size can be split by camel case, dots, or underlines into tokens.

Next, all words are reduced to their base form using lemmatization. Unlike stemming that simply chops off the ending of a word, lemmatization involves a complex word analysis and generally provides better results. Finally, we also remove repeated text sections, as quotations of the previous commentator in the bug report happen very frequently and the repeated text would affect the accuracy of text token distribution. Each bug tracking system may have their own mechanism to mark quotations. For example, Bugzilla based bug tracking systems, quotation starts with the greater sign (“>”) symbol on each new line and the quotation block has a CSS class of “quote”. Developers can design their own match patterns for removing quotations and plug it into PerfLearner.

After processing $o$ (the combined configuration option and its description), let $V$ be the vocabulary of all text tokens from both the bug report $br$ and $o$. Let $r = [wt,br | t \in V]$ and $o = [wt,o | t \in V]$ be the VSM representations of the bug report $br$ and the configuration option $o$. The term weights $wt,br$ and $wt,o$ are computed using the classical TF-IDF method described in existing literature [48]. After the vector space representations are computed, the textual similarity score between $o$ and $br$ can be calculated using the standard cosine similarity between their corresponding vectors:

$$sim(br, 0) = \cos(br, 0) = \frac{r \cdot o}{|r| \cdot |o|}$$

The score is computed by the inner product of the two vectors, divided by their Euclidean distance. For MySQL bug #74325 (Figure 6.1), by utilizing the configuration API description (Figure 6.7), PerfLearner ranks innodb_fill_factor at the top.

6.4.1.4 Identifying Workload

In performance testing, we need to know which frame elements are workload-sensitive, so testing can focus on generating workload values for these elements. We have defined six linguistic patterns to identify such frame elements. The most frequently used pattern is to locate
sentences containing benchmark tool names. Benchmark tools are often used to simulate workload in performance bug reports. For instance, MySQL bug report #74325 uses benchmark tool mysqlslap to generate a large number of database updates. Therefore, by searching for the benchmark name mysqlslap, we can detect that update is workload-sensitive. This pattern applies to 44.2% of performance bug reports involving specific workload.

The second commonly used linguistic pattern detects sentences describing workload information of data inputs (Figure 6.5). In this pattern, the data input (i.e. a text file) is followed by a verb (i.e. containing) that details the content of input data (i.e. a very long line). Once this pattern is detected, the corresponding data input is considered to be workload-sensitive.

**Figure 6.5 A Common Pattern to Determine A Workload**
6.4.2 Performance Test Frame Generation

PerfLearner generates performance test frames from the ranked frame elements, the workload specification, a strength file, and a constraint file. The strength file specifies top-N frame elements under each input category to be used for test frame generation. The constraint file is used to enforce constraints of interaction among frame elements, which can limit the number of (invalid) frames to be generated. The constraints are manually derived from user manuals. Both files are provided by users and generic to all bug reports within the same application.

Figure 6.6 shows a partial constraint file of MySQL. The data definition command DROP in the SQL works with DATABASE and TABLE but not with COLUMN. We use if to enforce conditions on which frame elements can be combined. To enforce the rule that UPDATE works with TABLE but not DATABASE, condition [if CMDUpdate] is added for data inputs COLUMN and

InnoDB performs a bulk load when creating or rebuilding indexes. This method of index creation is known as a "sorted index build". innodb_fill_factor defines the percentage of space on each B-tree page that is filled during a sorted index build, with the remaining space reserved for future index growth.
TABLE. Condition [if CMDDrop] is added for data inputs DATABASE. Therefore, when UPDATE is chosen, it can only be combined with COLUMN and TABLE. Our experiment indicates that adding constraints can reduce 70% of test frames.

In the example of Figure 6.6, the strength file indicates that top-2 commands ($n_c$), top-5% configuration options ($p_o$), and top-2 data inputs ($n_d$) are selected to generate test frames. Because the number of options is often large, we use a percentage of the total number of configuration options to indicate the selected number of configuration options. The three symbols $t_c$, $t_o$, and $t_d$ indicate the interaction strengths for commands, configuration options, and data inputs respectively. In Figure 6.6, a pairwise combination ($t_o=2$) is applied to the configuration options and no combinations are used for the command ($t_c=1$) and data inputs ($t_d=1$). Figure 6.6 also shows the default strength file used by PerfLearner. These strength values are chosen based on our empirical evaluation as they are the minimum requirements for generating test frames achieving up to 90% accuracy. We also evaluated the sensitivity of these values in Section 6.7.

Algorithm 2 describes the process of generating performance test frames. The algorithm takes as input a list of bug reports from an application, a strength file, and a constraint file. For each bug report, the algorithm obtains a ranked list for each input category (Lines 2-4) and a list of workload (Line 5). It then selects frame elements from the ranked lists with respect to the strengths. Next, a list of candidate test frames is generated given the selected frame elements and the constraints (Line 7). Finally, the algorithm ranks test frames collected from all bug reports (Line 10) in terms of the frequency of their appearance. Test frames ranked higher indicate they may be more likely to cause performance bugs. The last column of Figure 6.6 shows an example of the five test frames generated.

Algorithm 2 PerfLearner Test Frame Generation

<table>
<thead>
<tr>
<th>Algorithm 2</th>
<th>PerfLearner Test Frame Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires</td>
<td>StrtnF, ConstF, BugReports, LPts</td>
</tr>
<tr>
<td>Ensures</td>
<td>TF_prio</td>
</tr>
<tr>
<td>1</td>
<td>for $br \in$ BugReports do</td>
</tr>
<tr>
<td>2</td>
<td>RL_cmd $\leftarrow$ RankCmd ($br$, DL_cmd, LPts.cmd)</td>
</tr>
<tr>
<td>3</td>
<td>RL_o $\leftarrow$ RankConfig ($br$, DL_o)</td>
</tr>
<tr>
<td>4</td>
<td>RL_d $\leftarrow$ RankData ($br$, DL_d, LPts.d)</td>
</tr>
<tr>
<td>5</td>
<td>L_o $\leftarrow$ GetWorkload ($br$, DL_o, LPts.o)</td>
</tr>
<tr>
<td>6</td>
<td>RL $\leftarrow$ SelectElements (RL_cmd, RL_o, RL_d, StrtnF)</td>
</tr>
<tr>
<td>7</td>
<td>TF $\leftarrow$ GenerateFrames (RL, L_o, ConstF)</td>
</tr>
<tr>
<td>8</td>
<td>TF $\leftarrow$ TF $\cup$ TF</td>
</tr>
<tr>
<td>9</td>
<td>end for</td>
</tr>
<tr>
<td>10</td>
<td>TF_prio $\leftarrow$ RankFreq (TF)</td>
</tr>
</tbody>
</table>

6.4.3 Performance Test Case Generation

Algorithm 3 outlines the process of generating performance test cases from test frames. First, PerfLearner iteratively selects a test frame from the prioritized list output by Algorithm 2. For each frame element, the algorithm checks for its input category. If the frame element is workload-sensitive, depending on the input category, the algorithm applies workload in two
ways (Line 5). For the command category, the benchmark option that controls workload is included in the test case generation. For other input categories, the input size is included in the test case generation. The algorithm updates the test case as it gets more information from frame elements (Line 7). Specifically, the test frame is converted into an XML file (tc.xml) of which structure is known to the test case generation tools. Finally, the test input (tc.xml) is supplied to the performance testing tool. It is up to the performance testing tool to determine how to assign input values and execute the subject under test to detect performance bugs.

### Algorithm 3 Combining PerfLearner with Testing Tools

```plaintext
Require: TFrame
Ensures: TestResults
1: for f ∈ TFrame do
2: for e ∈ f do
3: category ← GetInputCategory(e)
4: if HasWorkload(e) then
5: c ← IntegrateWorkload(category, c)
6: end if
7: tce.xml ← UpdateTestCase(e)
8: end for
9: TestResults ← RunPerfTestTool(tc.xml)
10: end for
```

6.5 Implementation

We implemented a web crawler using the Python Beautiful Soup library [20] to collect raw bug reports and API documentations. We then leveraged Python Natural Language Toolkit (NLTK) [94] to parse the description of the bug reports and match linguistic patterns against the new bug reports with regular expressions on part-of-speech tags. For the information retrieval component, we utilized the Python machine learning library scikit-learn [122] to get the TF-IDF matrix and cosine similarity scores. Lastly, we implemented Python programs to handle the performance test frame generation.

6.6 Evaluation of PerfLearner

We evaluated PerfLearner on three open source projects with characteristics described in Section 6.3.1. We aim to answer the following research questions:

**RQ1:** How accurate is PerfLearner at detecting performance bug-triggering frame elements and workload?

**RQ2:** How effective and efficient is PerfLearner at generating performance test frames?

**RQ3:** Can PerfLearner enhance existing performance testing tools for detecting performance bugs?
6.6.1 Techniques and Metrics

RQ1: Accuracy of Bug Reports Analysis. To answer RQ1, we evaluate the accuracy of PerfLearner in extracting frame elements and workload. The techniques for extracting commands, configuration options, data inputs, and workload are denoted as CD, CO, DI, WL, respectively. Each technique is compared to a baseline method to evaluate the effects of using advanced techniques such as linguistic patterns and information retrieval (TF-IDF, Cosine Similarity etc.). Specifically, we compare CD, CO, DI to three baseline techniques – CD_s, CO_s, and DI_s. These baseline techniques use a keyword match and count the occurrence of each frame element appearing in a bug report. To evaluate the usefulness of configuration manuals in extracting configuration options, we also compare CO to CO_a. CO_a uses only tokens in the configuration option name without configuration manuals to make the similarity comparison. Since the workload describes whether a frame element is workload-sensitive, the keyword counting is not applicable in this case. Nevertheless, to evaluate the usefulness of linguistic patterns in identifying the workload, the baseline technique WL_r randomly selects a frame element and treats the element as workload-sensitive.

We use two metrics to evaluate the effectiveness of ranking. The first metric is the top-N success rate, which is computed by ranks of ground truths within top N items over all bug reports. For example, if 20 out of 100 performance bug reports rank the ground truth of configuration options in the top 5% of all 600 configuration options, the top-N (N=5%) success rate is 20%. When there are multiple elements specified as the ground truth, we only consider the first one that PerfLearner can find. Since workload is directly identified without ranking, we examine the percentage of bug reports in which ground truth workload is found.

For the second metric, we use MAP (Mean Average Precision). MAP is a single figure measure of ranked retrieval results independent of the size of the top list [121]. It is designed for general ranked retrieval problems, where a query can have multiple relevant documents. To compute MAP, it first calculates the average precision (AP) for each individual query Q_i, and then calculates the mean of APs on the set of queries Q:

\[
MAP = \frac{1}{|Q|} \sum_{Q_i \in Q} AP(Q_i)
\]

To illustrate the MAP calculation, suppose there are two configuration options o1 and o2 associated with a bug report. If Technique-I ranks the two options at the 1st and 2nd positions among all 500 options and Technique-II ranks the two options at the 1st and 3rd positions, then the MAP of Technique-I is \((1/1 + 2/2)/2 = 1\) and the MAP of Technique-II is \((1/1 + 2/3)/2 = 0.8\).
RQ2: Effectiveness and Efficiency of Generating Performance Test Frames. To answer RQ2, ideally, the comparison should be done with existing approaches that generate performance test frames. However, we cannot find an existing approach with this specific goal. In the absence of such approaches, we instead compare PerfLearner to a combinatorial testing (CT) strategy [55] that employs the category-partition method [97], t-wise testing [101], and the random testing approach. Specifically, CT generates test frames by combining elements under each input category with respect to the constraints. The first difference between PerfLearner and CT is that CT does not analyze bug reports or rank frame elements in terms of their relevance to the report; instead, CT ranks the frame elements in a random order. The second difference is that in CT, the workload is randomly assigned to a frame element. To make a fair comparison, the interaction strength of configuration options and that of data inputs are the same as those used in PerfLearner.

To evaluate the cost-effectiveness of PerfLearner and CT in generating performance test frames, we wish to know whether frame elements frequently appeared in historical bug reports can be used to generate test frame for testing future versions of the programs. For each bug report used for evaluation, we manually inspect and derive the test frame that triggers the performance bug described in the report (Section 6.3.1). We refer to this test frame as the ground truth test frame. Since test frames cannot be executed directly, we consider an approach detects the bug if the ground truth test frame is included in the generated test frames. To do this, we first list the 100 bug reports from each program in ascending order by the bug creation date. We then select the first 90 bug reports (training set) and apply techniques (PerfLearner and CT) described in Section 6.4.2 to generate test frames. We compare the test frames generated by each technique against the remaining 10 bug reports (test set) from each subject. Specifically, we examine at which iteration the ground truth test frame is generated by the technique. To evaluate the efficiency of the two techniques, we evaluate the time they take to generate the ground truth test frames.

RQ3: Detecting Performance Bugs. Besides evaluating PerfLearner on generating performance test frames, we would like to know whether the generated frames are useful for detecting actual performance bugs. PerfLearner is orthogonal to existing performance testing tools. It aims to improve the efficiency of testing by focusing on selecting commands and input parameters that are more likely to expose performance bugs. To answer RQ3, we combine PerfLearner with GA-Prof, a performance test input generation tool to detect performance bugs [125]. We choose GA-Prof because it is the only tool that can evolve both configuration option and data input values. GAProf employs a genetic algorithm to explore the space of input combinations among all input parameters. We re-implemented the genetic algorithm part of GA-Prof to handle C/C++ applications. We compare two settings of GA-Prof: 1) a default setting (denoted by GA) in which the combinations are evolved for all commands and input parameters,
and 2) an enhanced technique, denoted by GP.L, where it utilizes test frames generated from PerfLearner to iteratively select and evolve input values to generate performance test cases.

To evaluate whether the two techniques are able to detect performance bugs within a reasonable time limit, we select real performance bugs that we can reproduce. We iteratively select a bug report from the 1083 performance bug reports (excluding the 300 sampled bug reports in the dataset) and try to reproduce the bug. Because reproducing performance bugs is challenging and expensive, we stop this process after we have 10 bugs successfully reproduced – this process took approximately 400 work hours.

Next, we apply the two techniques to the program versions corresponding to the 10 performance bugs. We evaluate whether the performance bug described in the bug report can be detected and record the time it takes. Specifically, we conduct test experiments on High-Performance Computer (HPC) clusters. The basic HPC node is equipped with a 6 core 2.66 GHz Intel Xeon X5650 Westmere, 36 GB memory, and 256 GB hard drive. This environment enables us to run multiple experiments simultaneously without interruption. Each experiment is repeated 10 times and we report the mean to reduce the bias due to randomness. We default the time limit to 24 hours before terminating the experiment and set the maximum number of GA iterations in each run to be 10.

6.6.2 Results and Analysis

RQ1: Accuracy of Bug Reports Analysis. Table 6.3 shows the effectiveness of different techniques at ranking frame elements and extracting workload. The success rates are based on the default values specified in the strength file. The results indicate that commands appear in the top-2 positions for 82-91% of bug reports; the correct data input appears in the top-2 positions for 83-90% of the bug reports; the correct configuration option appears in the top-5% returned results for 71-85% of the reports. Additionally, the workload is identified with 56-80% accuracy. Compared to the baseline approaches, the success rate is higher in each category over all programs.

Where the MAP scores are concerned, PerfLearner is more effective than the baseline techniques over all three types of frame elements across all subject programs. The improvements range from 14% to 40%. These results suggest that heuristics used by PerfLearner is effective in boosting accuracy.

Table 6.3 RQ1: Test Frame Extraction Accuracy

<table>
<thead>
<tr>
<th>App.</th>
<th>Metric</th>
<th>Command</th>
<th>Data Input</th>
<th>Config. Option</th>
<th>Metric</th>
<th>Workload</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CD</td>
<td>CD_L</td>
<td>DI</td>
<td>CO</td>
<td>CO_L</td>
</tr>
<tr>
<td>Top-N</td>
<td></td>
<td>91%</td>
<td>78%</td>
<td>83%</td>
<td>67%</td>
<td>71%</td>
</tr>
<tr>
<td>MAP</td>
<td></td>
<td>0.80</td>
<td>0.70</td>
<td>0.70</td>
<td>0.60</td>
<td>0.73</td>
</tr>
</tbody>
</table>
RQ2: Effectiveness and Efficiency of Performance Test Frame Generation. Table 6.4 shows the results of PerfLearner and CT in generating performance test frames. Since CT does not rank test frames, we allow CT to generate test frames among randomly sampled input space for each input category. We limit the number of test frames to 10,000. The threshold number is based on practical considerations as 10,000 tests may take considerable executing time. With the default CT method, all three subjects failed to generate the ground truth test frame before the frame limit threshold. These results suggest that PerfLearner is more cost-effective at generating performance test frames than the traditional combinatorial testing approach. Figure 6.8 shows the distribution of test frames generated in each subject for both PerfLearner (PL) and CT. Firefox has the worst performance of all, this is largely due to Firefox bugs require multiple steps to trigger. Firefox also has the largest number of commands and lowest command extraction accuracy. As a result, the ranking of test frames does not work as effectively as the other two subjects.

<table>
<thead>
<tr>
<th>Application</th>
<th># of Const.</th>
<th>PerfLearner Space</th>
<th>PerfLearner Count Avg.</th>
<th>CT Space</th>
<th>CT Count Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>45</td>
<td>445K</td>
<td>2662</td>
<td>64M</td>
<td>10K</td>
</tr>
<tr>
<td>MySQL</td>
<td>12</td>
<td>1.4M</td>
<td>1831</td>
<td>2.9B</td>
<td>10K</td>
</tr>
</tbody>
</table>

Figure 6.8 Test Frame Generation
RQ3: Enhancing Performance Bug Detection. Table 6.5 shows the results of GA and GP_{PL} (GA enhanced with PerfLearner). GA failed to detect all 10 performance bugs. Like other test case generation techniques, the genetic algorithm for generating input values is applied only after a test frame is selected. However, without knowing which frame element is more likely to cause a performance bug, a random method is used to allow frame elements in each input category to have an equal chance to be selected. As a result, many low-quality test frames are generated. The ground truth test frame often fails to be generated within the time limit (24 hours).

Table 6.5 RQ3: Performance Testing with GA

<table>
<thead>
<tr>
<th>Application</th>
<th>Bug ID</th>
<th>Effectiveness</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GA</td>
<td>GA_{PL}</td>
</tr>
<tr>
<td>Apache</td>
<td>54852</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>52914</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>37680</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>43081</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>46749</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>MySQL</td>
<td>21727</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>44723</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>74325</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>15653</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>26938</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

Our results show that the GP_{PL} approach can detect seven out of 10 performance bugs within an average of 10.9 hours. These results suggest that PerfLearner can potentially enhance existing performance testing tools. For the three bugs GP_{PL} failed to detect: 1) Apache bug #37680 requires two entries of the “Listen” option. When selecting configuration options, we do not allow duplications of configuration option since multiple appearances of the same option normally overwrites one another. 2) Apache bug #46749 executes a test frame (server graceful stop) that causes a long response time. This test frame is considered to trigger a performance bug, however, the ground truth test frame of this bug is related to cache utilization. This is the only false positive case appeared in our experiment. 3) For MySQL bug #26938, the “profile” command is required to trigger this bug. However, none of the bug reports used to generate test frames includes the command “profile”. We conjecture false negative cases can be reduced as more bug reports are used for mining test frames.
6.7 Discussion

**Sensitivity of Strength.** By default, PerfLearner uses strengths \{ \( n_c = 2, n_o = 5\% , n_d = 2, n_w = 2, t = 2 \) \}. The selected values are based on the empirical study that achieves best test frame element extracting results. To understand the influence of selecting different sets of strengths, we evaluate PerfLearner on two other sets of strengths: \( w_1 = \{ n_c = 1, n_o = 2\% , n_d = 1, n_w = 1, and \ t = 1 \} \) and \( w_2 = \{ n_c = 3, n_o = 10\% , n_d = 3, n_w = 3, and \ t = 3 \} \). Figure 6.9 reports the results of test frame generation using the three sets of strengths on the test set (10 bug reports) for each of the three subjects. The vertical axis indicates the number of frames generated before reaching the ground truth. The results indicate that, in general, default strengths outperform the other two sets. In Apache, \( w_1 \) outperforms the default strengths in terms of the average frames generated, but \( w_1 \) exhibits a larger standard deviation. The weight sensitivity analysis implies that the strengths should not be set too low or too high. Low strength values may cause PerfLearner to miss certain relevant frames, whereas high strength values may result in generating too many performance test frames and thus reduce the efficiency of PerfLearner.

**Threats to Validity.** The primary threat to the external validity of this study involves the representativeness of our subjects and bug reports. We do reduce this threat to some extent by using several varieties of well-studied open source projects and bug tracking systems for our study. Combining keyword search and manual inspection is an effective technique to identify bugs of a specific type from a large pool of generic bugs and has been used successfully in prior studies [72, 92, 160]. We cannot claim that our results can be generalized to all systems of all domains though. The primary threat to the internal validity involves the manual inspection to
identify the ground truth test frame from a bug report. To minimize the risk of incorrect results given by manual inspection, the analysis process was done independently by two trained inspectors.

**Limitations.** The textual quality of a bug report has substantial impact on the effectiveness of the proposed approach. For example, a bug report may not use the standard names of the frame elements. This can be addressed by integrating advanced NLP techniques, such as Word2Vec [156]. The incompleteness of bug reports is also a major obstacle for PerfLearner to work well, like for many bug report analysis techniques. One strategy is to filter out bug reports containing missing information using an automated approach [30] and apply PerfLearner only to complete bug reports to improve accuracy. Other classification techniques can be integrated with PerfLearner as well, such as detecting reproducible [47] and duplicate [68] bug reports.

PerfLearner takes only labeled performance bug reports. One extension point is to build a prediction model that can automatically predict whether a new bug report is related to performance or not. There has been some research on using text mining to classify bug reports [51, 148, 154], which can be easily tuned to handle performance bug reports. In addition, when a performance bug requires a specific system state (e.g., networking events) to be triggered, the current approach cannot extract such information. For example, a state may be associated with the topology of the target system (e.g., the firewall setup may negatively affect the performance of a system). Nevertheless, we believe PerfLearner can be extended to handle system-level triggering events by defining additional frame databases and linguistic patterns.

6.8 Related Work

There has been a great deal of research on analyzing, detecting, and fixing performance bugs [27, 72, 73, 93, 96]. Burnim et al. [27] designed a technique to generate worse-case inputs (larger input sizes) to find performance bugs. As discussed in Section 6.1, these techniques often rely on initial test cases and do not address the challenges of finding the right combination of input parameters to create effective initial test cases. As our empirical study shows, workload only helps to trigger some but not all performance bugs. Although PerfLearner also takes workload into consideration, it focuses more on the combination of elements to be used in the test frame. Our method is orthogonal to the test case generation tools, as our experiment shows, PerfLearner can be integrated into existing performance testing techniques to improve the effectiveness and efficiency of bug detection.

A great body of work has been conducted on applying combinatorial testing (CT) to address the problem of large input space in complex and configurable systems [43, 90, 158, 167]. CT systematically samples the input space and tests only the selected input parameters combinations. Zhang et al. [167] proposed a method to optimize combinatorial testing to
generate test cases to find a balanced point of coverage without pressuring on achieving the maximum coverage. Dumlu et al. [43] proposed a feedback-driven approach to detect and avoid masking effect resulting from CT. These techniques focus on sampling combinations from the entire input space. Therefore, it is often inevitable to result in a large sampling space. In the contrast, PerfLearner detects and uses only the error-prone commands and input parameters from the historical bug reports. Empirical results show that our approach can significantly reduce the sampling space when generating test frames for performance bugs.

There has been considerable work on using natural language and information retrieval techniques to improve code documentation and understanding [30, 46, 64, 65] and to create code traceability links [7, 42, 99]. While our work applies some of these same basic techniques, such as tokenization, lemmatization, vector space model with term frequency-inverse document frequency weighting [19], the prior work has not applied these techniques to performance bug reports and has not considered or extracted input parameters to generate test frames.

There has been a large body of work that demonstrates the need for configuration-aware testing techniques and proposes methods to sample and prioritize the configuration space [69, 89, 111, 119, 159] to reduce the cost of testing. For example, Jamshidi et al. [69] conduct an empirical study to evaluate the feasibility of applying the transfer learning technique to reduce the dimensionality of the configuration space when constructing performance models. Nair et al. [89] use inexpensive and inaccurate models to find optimal configurations with less cost compared to the state-of-the-art sampling techniques. Unlike the above technique, our approach focuses on creating test frames to aim performance testing for finding performance bugs instead of performance modeling.

6.9 Conclusions

Performance bugs are difficult to expose because they often manifest under special input conditions and system configurations. In this paper, we studied 300 real-world performance bugs from three popular open source projects. Our findings indicate that combinations of input parameters, especially configurations, can play an important role in exposing performance bugs. Guided by these findings, we designed PerfLearner, an automated approach to extract test frame elements, and to generate test frames for performance testing. We evaluated PerfLearner on 300 bug reports and the results show that PerfLearner extracts test frame elements with high accuracy. PerfLearner is also effective in generating performance-bug-triggering test frames. Our evaluation on combining PerfLearner with GA-Prof to detect real-world performance bugs indicates that PerfLearner can enhance existing performance testing tools for generating test cases and detecting performance bugs. For reproducibility and further
research, PerfLearner and all the data from the experiments are publicly available at https://github.com/xha225/PerfLearner.
Modern computer systems are highly-configurable and the misconfigurations can easily lead to performance issues. However, the sheer size of the configuration space makes it challenging to manually adjust the software to fix the issues and achieve long-term performance gains. In this chapter, we propose an automatic performance tuning approach, ConfRL, that can automatically select and tune configurations in response to environment dynamics to optimize the system’s performance. The key idea of ConfRL is to use reinforcement learning that enables an agent to learn in production runtime environment by trial-and-error and use the feedback to tune configuration options for achieving better performance. To reduce the cost of learning in the presence of large configuration space, ConfRL employs sampling, clustering, and dynamic state reduction techniques to reduce the states needed for reinforcement. Our evaluation on four real-world highly-configurable open-source projects showed that ConfRL can efficiently and effectively guide software systems to achieve higher long-term performance gains.

7.1 Introduction

Modern computer systems are highly-configurable, allowing users to customize a large number of configuration options to meet their specific goals. The complexity of the configuration space and the sophisticated constraints among configuration settings could easily lead to performance issues. Recent studies have shown that performance problems caused by misconfiguration are still prevalent [17, 63, 72]. Unlike functional bugs that typically lead to system crashes or incorrect results, a performance issue can cause significant performance degradation which leads to long response time and a low program throughput [26, 72, 92].

When a performance problem occurs (e.g. a significant slowdown with HTTP responses in a web server), system administrators and developers may need to reconfigure the system to find a configuration setting for better performance. However, it is often not an easy task to figure out the best settings for a system with a large number of configuration options. For example, the latest version of Apache HTTP Server has 618 configuration options. For a developer with limited domain knowledge, it is difficult to pinpoint the problem and find an optimal performance setting.

Even for domain experts, it is often not an easy task to configure the software system to get the best performance. For example, as one experienced user complained in Apache HBase bug #13919, “There are current many settings that influence how/when an HBase client times out. This is hard to configure, hard to understand, and badly documented.” In addition, manually changing the configuration can be tedious, inefficient, and impractical. For instance, in the case of a web server, the volume of the request level changes at different times of the day. It is not
practical to ask administrators to change configuration settings to keep up with the level of web request changes.

The goal of this research is to develop an approach, ConfRL, that can automatically select and tune configuration options in response to the environment dynamics to achieve higher performance. ConfRL is intended to be used by system administrators and developers to tune performance affected by configurations. The key idea of ConfRL is to use reinforcement learning (RL) techniques to automate performance configuration tuning. RL is a process of interacting with a dynamic environment to generate the optimal control policy on what actions to take for a given state. Therefore, we can formulate the task of tuning performance configuration as a RL problem, in which the optimal policy refers to a configuration generated for achieving higher performance with respect to the current system state. The main benefit of RL is that it does not require domain knowledge of the system and is able to update optimal policies continuously in the long run.

ConfRL consists of two stages: performance-influential configuration option ranking and Q-Learning [153]. The first stage identifies configuration options that potentially influence the system’s performance. Since the enormous configuration space often leads to a huge number of states that RL must explore, apply RL directly to a system with a large number of configuration options hardly scales. To address this challenge, ConfRL uses a clustering method to identify performance-influential options.

The second stage uses Q-Learning for finding a policy to provide guidance on what actions to take on a given state to achieve higher performance gains. There are two challenges in this stage. First, even with fewer options (identified by the first stage) to choose from, the number of configurations may still be enormous [144] at runtime. To address this challenge, we utilize adaptive value generation and dynamic state merging techniques to reduce the runtime reinforcement learning states. Another challenge is the inconsistent readings of the performance measurements (e.g., program throughput such as requests per second) across multiple system executions with exact same inputs and configurations. Such inconsistencies make it difficult to determine if one set of configuration is truly better than the other with respect to performance gains and thereby may disturb the results of the calculated reward values in Q-learning. We address this challenge by developing performance measurement caching techniques so performance measurements for any visited state are cached to obtain consistent performance readings.

ConfRL differs from existing work [70, 137] on configuration tuning for correcting issues. For example, PrefFinder [70] uses information retrieval (IR) from static documentation to automatically find user preferences for correcting the configuration of a running system. In contrast, ConfRL focuses on addressing performance problems and does not assume the
availability of any documentation, which may be incomplete or out of date. Swanson et al. [137] propose REFRACT, a self-adaptive approach to avoid software failures. In comparison to this work, our goal is to achieve higher long-term performance gains, which is intrinsically different than software failures avoidance. Second, sampling may suffer from a scalability problem as the number of options increases. More importantly, sampling by itself does not provide a policy to guide on what actions to take. Instead of relying on sampling techniques to find workarounds, we use reinforcement learning that takes advantages of past interactions between the agent and the environment to guide the subject systems toward better software performance.

We evaluate ConfRL on four popular real-world C/C++ programs. Our results show that ConfRL is effective in improving performance through automated configuration tuning in a reasonable time period. Compared to a random tuning approach, ConfRL is up to 30% more effective in achieving performance gains. Moreover, the optimization techniques employed by ConfRL significantly reduce the number of states for reinforcement learning up to 82.5% and thus require less time (20.5 hours) to converge.

In summary, our paper makes the following contributions:

- An automated tool that directs system configurations to achieve long-term performance gains.
- An approach that uses reinforcement learning to automate performance configuration tuning and a set of optimization techniques for scalability and adaptability.
- A practical implementation and empirical evidence to show that the approach can effectively and efficiently improve performance in real-world server programs.

7.2 Background

In this section, we introduce the background of reinforcement learning (RL) and discuss how to model automated performance tuning as an RL task.

7.2.1 Tuning Performance Configuration

Reinforcement Learning (RL) [116], as illustrated in Figure 7.1, is the procedure of learning from interactions between an agent and the environment to determine the best action to take under any given state to achieve the maximum long-term rewards [136]. The agent first initiates an action. The environment reacts to the action by transiting the agent to a different state. Based on the current and previous states, the environment calculates what rewards to grant to the agent. This cycle goes on iteratively until the learning procedure terminates. The output of RL is a policy that maps the agent’s current state to the best action it should take. The value of an action in a state is used to indicate how good the action is in that state, which is computed by
a function that estimates the sum of future rewards by taking this action. The agent performs trial-and-error interactions with the environment to obtain the reward. Therefore, the optimal policy is to select the action that maximizes the reward in each state.

Figure 7.1  Weight Sensitivity Analysis

The task of tuning performance configuration can fit into the RL framework. An agent is responsible for initiating a change to configurations. A state reflects the system configuration (i.e., a combination of configuration option values). Each time the agent tunes (i.e., through an action) the configuration, it receives a reward in the form of performance measurements. After sufficient interactions, the agent obtains an estimate of how good an action is for the current configuration (i.e., state). Therefore, given a new state, the agent is able to tune the optimal configuration by matching the current state to a known state and take the action with the highest performance reward.

7.2.2  Reinforcement Learning Techniques

Depending on what is available to the problem (e.g. full knowledge of the environment such as the transition function), reinforcement learning comes in a few different forms. We discuss briefly two of the most widely adopted methods and explain why we select the method that is suitable for our problem.

Markov Decision Process. The basic form of a reinforcement learning problem is encapsulated as the Markov Decision Process (MDP) [110]. Formally, an MDP is used to describe an environment for reinforcement learning, where the environment is fully observable. The Markov Decision Process consists of a finite set of states, a finite set of actions, a state transition matrix: \( P_{ss'} = P[S_{t+1} = s' | S_t = s] \) [115], a reward function \( R \), and a discount factor \( \gamma \). A state has Markov property if and only if each state captures the information from all past states that lead to the current state. A policy \( \pi \) gives the probability to take an action given a state \( s \): \( \pi(a | s) = \ldots \)
The action-value function \( q_\pi(s, \alpha) \) is the expected total rewards given state \( s \) by taking an action \( \alpha \) following the policy \( \pi \). The goal of solving the MDP problem is to find the optimal action-value function: \( q_\pi(s, \alpha) = \max_{\pi} q_\pi(s, \alpha) \).

**Model-Free RL.** If a problem can be modeled as MDP, it can be solved analytically through value-iteration and policy-iteration algorithms. However, for problems having a large number of states (e.g. the state space in the millions), the inversion of the state transition matrix can be very expensive. Most real-world problems cannot be formulated as MDP since the environment is not fully observable. It is also difficult to describe the rules in a dynamic environment, so the MDP transition function is unknown. There are a set of techniques to estimate the action-value function of an unknown MDP, such methods are referred to as Model-Free reinforcement learning algorithms [115]. The Q-Learning method [153] is one type of Model-Free learning. It seeks to learn a policy to maximize the total award which naturally fits our problem. Q-Learning is based on the Bellman Optimality Equation [22]:

\[
q_\pi(s, \alpha) = R_s^\alpha + \gamma \sum_{s' \in S} \max_{a'} q_\pi(s', a').
\]

ConfRL uses Q-Learning as its reinforcement learning algorithm. The formula is made up of two parts: \( R_s^\alpha \) is the immediate reward, \( \sum_{s' \in S} \max_{a'} q_\pi(s', a') \) is the expected future reward to take action \( \alpha \) in state \( s \), and \( \gamma \) is the discount factor that determines how valuable the future reward is.

**Epsilon-Greedy Exploration.** As the agent explores the environment, it takes advantage of the past experience interacting with the environment. Internally, the agent maintains a state-action table that keeps track of the reward received by taking specific actions in a state. This helps the agent to find a path that leads to a higher performance gain. However, in the early stage of exploration, sticking to the state-action table completely may restrict the number of states the agent could have visited. There may be a chance that the agent could have achieved higher performance by visiting different states. By convention, the degree to which the agent acts randomly is denoted as epsilon. Epsilon has a range between 0 (i.e. no random actions at all) and 1.

### 7.3 Problem Formulation

We demonstrate how to formulate and solve the problem of performance tuning by RL using an example of the Apache web server. Column “Option Name” in Table 7.1 lists the names of the selected configuration options. Column “Type” lists the option types: binary (B) or numerical (N). Column “Range” lists the configuration option value range. Column “Constraints”
lists constraints of imposed on configuration options. The value range and constraints are manually extracted from the documentation of the application and are saved into a file.

Table 7.1 Configuration Option

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Range</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeepAlive</td>
<td>B</td>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>MaxClients</td>
<td>N</td>
<td>[1,512]</td>
<td>MaxClients &lt; ServerLimit</td>
</tr>
<tr>
<td>StartServers</td>
<td>N</td>
<td>[1,100]</td>
<td>StartServers &lt; MaxSpareThreads</td>
</tr>
<tr>
<td>ThreadsPerChild</td>
<td>N</td>
<td>[1,128]</td>
<td>ThreadsPerChild * StartServers &lt; MaxClients</td>
</tr>
</tbody>
</table>

Figure 7.2 ConfRL Overview

State. A state is encoded as an instance of an application’s configuration settings. For example, Table 7.2 illustrates five states in Apache. Each state is a combination of configuration option values (Columns “Option”) currently being used. The “I.I.D.” column lists the interaction ID, which is used to keep track of the number of interactions between the agent and the environment. The “S.I.D.” column lists the state ID. The default setting of configuration options is used as the initial state (S1). We discuss the impact of choosing different initial states in Section 7.7. The “Measure.” column lists the performance measurements. It is a measure to quantify software performance. For web servers, we measure the number of concurrent web page requests per second [1]; for database servers, we measure the number of transactions per second [37]. The “Action” column lists the next selected actions to be performed on the subject. The “Rewards” column lists the immediate performance reward. The reward is used to populate the state-action table, a.k.a., the Q-Table. The “State” column in Table 7.3 lists the visited reinforcement learning states. Column “A1” to “A8” list the eight actions associated with the four options used in Table 7.2. Each cell in Table 7.3 lists the immediate performance reward by taking an action in the corresponding state. By default, the performance reward is set to 0.
Table 7.2 Apache RL States

<table>
<thead>
<tr>
<th>I.ID</th>
<th>S.ID</th>
<th>Options</th>
<th>R/S</th>
<th>Action</th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>K.A.</td>
<td>M.C</td>
<td>S.S.</td>
<td>T.P.C</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
<td>OFF</td>
<td>102</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
<td>ON</td>
<td>102</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>S3</td>
<td>ON</td>
<td>102</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>S4</td>
<td>ON</td>
<td>256</td>
<td>32</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 7.3 State-Action Table

<table>
<thead>
<tr>
<th>State</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.25</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.17</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Action.** An action is an update issued by ConfRL to modify an individual configuration option value. For numerical option types, an action can be 1) increasing an option value; 2) decreasing an option value. For binary options types, an action can be 1) setting a binary option value to True; 2) setting a binary option value to False. Action is indexed and encoded by an integer number. Each integer is mapped to a specific operation on the option, as shown in the “Action” column of Table 7.2. In this example, action one (A1) is mapped to setting the first configuration option, KeepAlive, to “ON”. Action five (A5) is mapped to increasing the value of the third configuration option (i.e., StartServers).

**Reward.** A reward is calculated based on performance measurement. For example, the performance measurement for a web server can be the number of concurrent web page requests per second and that for a database server measured by the number of transactions per second. ConfRL first obtains the performance measurement under the current configuration options (M_C). It then calculates the immediate reward by the following rules: if the agent enters a state that leads to better performance (M_N), the environment assigns the agent with a positive reward; otherwise, the environment punishes the agent by assigning a negative reward. The reward is the relative difference between M_N and M_C, and the normalization puts options with a large value range on the same scale: Rewards = (M_N - M_C) / M_C. Since obtaining the performance measurement for each execution is time-consuming and that the measurements may be inconsistent due to environment dynamics, ConfRL builds a cache between a state and its performance measurement. The cache can 1) speed up the process of getting performance measurement, and 2) guarantee the performance measurement is consistent throughout the learning process. The details of the performance caching technique will be discussed in Section 7.4.5.
7.4 Design of ConfRL

Figure 7.2 gives an overview of ConfRL. The input to the approach is a configurable program, its associated configuration options, and their constraints. In the configuration option selection phase, ConfRL first selects options that influence the system’s performance — the goal is to reduce the states and thus the cost of learning. ConfRL employs a sampling and a clustering technique to rank the performance-influential options and assign them with appropriate weights. ConfRL then uses an adaptive value generation method to systematically generate adequate numbers of option values to cover a wide performance value range. ConfRL calculates the reward based on the performance measurement obtained in each state and uses the reward to build the Q-Table.

To reduce the cost of learning, ConfRL employs a state merging method to merge reinforcement learning runtime states that share the same performance measures. This can effectively reduce the size of the state space and a smaller state space leads to faster learning. In the end, ConfRL outputs a Q-Table to reflect the latest interactions between the agent and the environment. This procedure goes on until when the iteration threshold (i.e., stopping criterion) has been reached. For instance, the learning procedure stops after 24 hours in our case.

7.4.1 The ConfRL Algorithm

Algorithm 4 illustrates the pseudocode of ConfRL. The algorithm takes as input the subject’s configuration options with default values and outputs a Q-Table. The algorithm starts with a $t$-way sampling technique [159] to get the clustering training data (Line 1). Then, the ranking of performance-influential options is generated from clusters (Line 2). ConfRL creates an action-value table to store the calculated rewards (Line 3). Next, the algorithm initiates the Q-Learning algorithm (Line 4). There are two hyper-parameters involved in the Q-Learning algorithm: the learning rate (alpha) and the discount factor (gamma). The learning rate “determines to what extent newly acquired information overrides old information” [153], which controls how fast the reinforcement learning converges. The discount factor weighs on how the learning agent perceives for future rewards. The discount factor has a value between 0 and 1, where 0 indicates the agent takes only the immediate rewards without considering for a long-term reward (i.e. the expected reward for taking an action onwards) and 1 indicates the agent favors the learning towards long-term rewards.

The $\epsilon$-Greedy explorer (Line 6) is used to control the degree to which the agent follows the original policy. It is a discrete explorer that follows the greedy policy while maintaining a chance to take a random action to explore the unknown states. The algorithm instantiates the environment and the agent objects (Lines 7–8,) and then starts the reinforcement iterations
Inside the while loop (Line 9). The agent is reset to the original state at the beginning of each iteration to start from the initial state and exploit what the agent has learned from the environment. When the iteration starts, the DoInteractions function (Line 10) controls interactions between the agent and the environment. This is also where adaptive value generation starts. The algorithm dynamically adjusts the value of epsilon (Lines 11–13), with the goal of reducing the value of epsilon as the agent has more interactions with the environment to converge faster. State merging (Line 15) is used at the end of the learning iteration to reduce the number of runtime states. The learning process continues until it reaches a time-threshold (Line 7) to terminate.

```
Algorithm 4 ConfRL

Require: Initial set of configuration options, O
Ensure: Q-Table
1: trainingData ← tWaySampling(O)
2: rankedOps = PerfClustering(trainingData)
3: act ← ActionValueTable()
4: learner = QLearner(alpha, gamma)
5: explorer = EpsilonGreedyExplorer(epsilon)
6: learner.SetExplorer(explorer)
7: env = ConfigLearnEnv()
8: agent = LearningAgent(act, learner, env)
9: while ReachTimeLimit <> TRUE do
10:   DoInteractions()
11:   if TriggerEpsilonUpdate() == TRUE then
12:     epsilon = UpdateEpsilon()
13:   end if
14:   env.StateMerging()
15: end while
16: function StateMerging
17:   perfM = MeasurePerf()
18:   if perfM ∈ PerfState.keys() then
19:     masterState = PerfState[perfM]
20:     UpdatePerfState(masterState, stateID)
21:   else
22:     PerfState.add(perfM, stateID)
23: end if
24: end function
```

7.4.2 Ranking Performance-Influential Configurations

This pre-processing step is used to identify the performance-influencing configuration options from the target option space. A $t$-way covering array samples the set of configurations in such a way that all possible $t$-way combinations of options appear at least once. Based on a previous study [36], a 3-way covering array is adequate to cover 90% of interactions between options for configuration sampling. Configuration options are grouped based on their functional units (a.k.a. modules), for instance, configuration options under the core and mpm_common modules are grouped in Apache for testing out web page requests.

A 3-way covering array is used to conduct configuration sampling. The sampled configuration options are executed and the performance measurements are stored. The
intuition is that performance-influential options tend to influence performance through drastic value changes. The configuration option often goes beyond a threshold value to change program performance significantly. For instance, in Apache bug #54852, only when the StartServers option is set to a relatively large number (i.e. StartServers = 64) will it cause a slowdown in Apache. Clustering method naturally distinguishes such options by putting data instances in the corresponding clusters.

The popular K-Means [75] clustering method is used for its ease of interpretation and implementation. The clusters with the highest and lowest mean performance measurement are selected. Because the performance measurement is a direct result of the configuration options, options in each cluster may be used to describe the characteristics of the underlying clusters. Therefore, the mean value of each configuration option is used to calculate the difference for both data set. We measure the difference in option value changes. And the configuration options are ranked in a descending order based on the difference. Because only a small subset of configuration options can lead to performance degradations [63], the top-10 of the ranked options get a higher weight in the reinforcement learning to have a higher chance to be selected.

7.4.3 Generating Option Values

Extensively selecting all the legal values within the configuration option value range is not practical. In this step, we want to select option values that cover a decent value range without exhaustively try out every possible value. The value range for each configuration option is extracted from the subject documentation in the form of [OPT_MIN, OPT_MAX]. OPT_MAX is set to two times the size of the recommended max value to ensure a wide value range coverage to expose performance problems [155].

ConfRL utilizes an adaptive value generation strategy to generate option values. Unlike the sampling method where data points are calculated before any execution. RL is a dynamic process which requires the option values to be generated on the fly with respect to the action. Initially, each option starts with the default value. Because the impact of configuration option values on the subject performance is not continuous, performance bugs are perceivable by going above or below a threshold value [73]. Therefore, it is not necessary to generate option values in a continuous manner. Similar to the binary search algorithm, the speed of the value adjustment is changed by a factor of two. Therefore, the maximum number of option value choices is bounded by log(OPT_MAX). For instance, the MaxClients option has a value range of [1, 1024]. As such, we have at most eleven values choices from for MaxClients, namely, 20, 21, ..., 29, 210. ConfRL makes sure the selected value satisfies all constraints imposed on the option using the python-constraint library [91].
7.4.4 Reducing Runtime States

The adaptive value generation strategy can significantly reduce the number of individual option values to be used to cover a wide value range suitable for exposing performance problems. However, as the number of configuration options used in the learning process increases, it still poses a challenge to handle a large number of option value combinations, aka, the reinforcement learning states. To further reduce the number of reinforcement learning states, ConfRL uses a dynamic state reduction strategy.

At runtime, we notice that different states do not always lead to different performance. Only those configuration options that have a performance impact tend to lead to different performance measurements. The redundant states lead to unnecessary cost in measuring performance without providing any new insights for the reinforcement learning process. As such, ConfRL merges reinforcement learning states that share the same runtime performance. ConfRL implements a cache to store performance measurements for states. At the end of each reinforcement learning iteration (Line 15 in Algorithm 4), a reference list is constructed for state IDs that have identical performance measurements (Lines 21–22 in Algorithm 4). The first such state in each reference list is referred to as the master state, the rest of the states in the reference list is referred to as slave states. In the following reinforcement learning iteration, ConfRL returns the master state if the current state is in the slave state list.

7.4.5 Measuring Performance

Performance measurements (e.g., execution times, throughputs) are used to evaluate if one state is better than another state in terms of achieving higher performance. Performance measurement is essential to calculate rewards in ConfRL for Q-Learning (Section 7.3). Within each reinforcement learning iteration, ConfRL measures the performance by executing benchmark tools (e.g. Apache Benchmark, DBT-2). Due to the changing dynamics of the environment, performance for the same state may vary from time to time. The inconsistency in the performance measurement has a negative effect on the reward calculation, therefore impacting the agent’s decision on choosing the best action to take in a given state.

To provide a reliable and consistent performance measurement, the performance measurement of a state is store upon the first time ConfRL explores the state. Specifically, the state and its performance measurement are stored in a dictionary. The dictionary uses the state ID as the key and the corresponding performance as its value. This dictionary serves as a performance measurement cache for states. In each subsequent reinforcement learning iteration, the performance of the same state is queried and retrieved directly from the cache instead of re-running the benchmark utility. This strategy guarantees the performance of the
same state is consistent throughout the learning process. It also reduces the overall exploration
time as benchmark tools can take a significant amount of time to calculate the performance
measurement.

7.4.6 A RL Running Example

To demonstrate the reinforcement learning design of ConfRL, we use Table 7.2 to illustrate
the way ConfRL works. Table 7.3 illustrates the status of Q-Table as the learning progresses. In
this example, we assume the StartServers option has a higher weight than other options.
Therefore, the option has a higher chance to be explored by the agent (e.g. the StartServers
option has been selected in two out of five cases).

The process starts at the state $S_1$: $\{\text{OFF,102,12,3}\}$. The state $S_1$ has a performance
measurement of 10 requests/second (r/s). In the first interaction, the agent receives the action
1, that is to modify the value of the KeepAlive (K.A.) option. ConfRL looks up the configuration
option type and confirms that KeepAlive is a binary option type. ConfRL assigns the value ON to
KeepAlive, as such, Apache is now in a new state, $S_2$: $\{\text{ON,102,12,3}\}$. After running the
benchmark tool, Apache gets a performance measurement of 20 r/s. In the meanwhile, ConfRL
calculates the immediate reward for taking action 1 (A1) in $S_1$ is 1 (i.e. (20-10)/10). Table 7.3
gets updated to keep track of the rewards assigned in the state $S_1$.

In the second interaction, the agent receives the action 5 (A5) to modify the StartServers
(S.S.) option. ConfRL recognizes StartServers as a numerical option type and calculates the next
option value for StartServers. Besides taking the option’s type and its current value (i.e. S.S. =
12), ConfRL checks the option constraint to make sure that all constraints associated with this
option are still intact (e.g. ThreadsPerChild * StartServers < MaxClients must hold true for
StartServers). To increase the value of StartServers by two, we get 24. The adaptive value
generation adjusts the option value by finding the first value in the series (i.e. $2^n$) that is larger
than 24. Therefore, the StartServers option gets a value of 32. Apache is now in the state $S_3$:
$\{\text{ON,102,32,3}\}$. The performance measurement of $S_3$ is 25 r/s. Therefore, the immediate reward
for taking action 5 in $S_2$ is 0.25.

In the third interaction, the agent receives the action 7 (A7) which is to increase the value
of ThreadsPerChild (T.P.C.). ThreadsPerChild gets a value of 4. When the ConfRL validates the
constraints, it no longer holds: ThreadsPerChild (4) * StartServers (32) > MaxClients (102).
ConfRL uses the Constraint Satisfaction Problems (CSPs) solver. The constraint is passed to the
One solution to satisfy the constraint is $\{\text{T.P.C.:4, S.S.:32, M.C.:256}\}$. As such, ConfRL assigns 256
to MaxClients. Apache is now in the state $S_4$: $\{\text{ON,256,32,4}\}$ with a performance measurement
of 30 r/s. The immediate reward for taking action 7 in $S_3$ is 0.2.
In the fourth interaction, the agent receives the action 6 (A6), which is to decrease the value of StartServers (S.S.). The StartServers option gets a new option value of 16. The subject is now in the state S5: {ON,256,16,4} with performance measurement of 30 r/s. The immediate reward for taking action 6 in S4 is -0.17. After the first iteration finishes, dynamic state reduction looks through states that have identical performance and combines such states. For instance, states S2 and S5 will be combined, and the performance-state dictionary gets a new entry: {20: {MasterState: S2, SlaveStates: S5}. As the learning iteration advances, the Q-Table gets populated and updated to allow the best action to be returned based on the current state.

7.5 Empirical Study

To evaluate ConfRL, we conduct an empirical study on four subjects and aim to answer the following research questions:

**RQ1**: How effective is ConfRL in tuning the values of configuration options for achieving long-term performance gains?

**RQ2**: How efficient is ConfRL for achieving a given performance goal?

7.5.1 Implementation

For reinforcement learning, we extend the Python-based library pybrain [120] to conduct Q-Learning. The Bash shell script is used as a driver to conduct various experiments. We conduct all the experiments under Red Hat Linux on a High-Performance Computer (HPC) cluster. The basic HPC node is equipped with a 6 core 2.66 GHz Intel Xeon X5650 Westmere, 36 GB memory, and 256 GB hard drive.

7.5.2 Subject Programs

We choose four popular open-source server applications: Apache HTTPD Server, Lighttpd Web Server, MySQL Server, and PostgreSQL (PSQL) Server. All subjects are highly configurable server applications, which are prone to performance issues caused by misconfigurations. Table 7.4 shows the characteristics of the subjects. The “Module” column shows the modules from which the configuration options are collected. We evaluated the modules involving the core functionalities of the programs. The “#$On$” column lists the number of numeric options and the “#$Ob$” column lists the number of binary options.

To evaluate the performance of subject programs, we choose the program throughput as the performance measurement similar to other work [25]. Specifically, we use the concurrent HTTP request (CHR) for web servers and the number of transactions per second (TPS) for
database servers. CHR and TPS are commonly used performance measurements for web and database servers.

Table 7.4 Characteristics of Subjects

<table>
<thead>
<tr>
<th>Subject</th>
<th>Modules</th>
<th>#O₁</th>
<th>#O₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>CORE, WORKER, MPM_COMMON</td>
<td>33</td>
<td>17</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>CORE</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>MySQL</td>
<td>INNODB, SYSTEM</td>
<td>29</td>
<td>24</td>
</tr>
<tr>
<td>PSQL</td>
<td>CORE</td>
<td>40</td>
<td>38</td>
</tr>
</tbody>
</table>

7.5.3 Experiment Design

7.5.3.1 Baseline Techniques

We use a random method MRND as the baseline for both effectiveness and efficiency comparison since there are no existing techniques that can achieve the same goal as ConfRL. MRND randomly selects a configuration option from the configuration option pool and then assigns a random value to the configuration option according to its value range. MRND skips state reduction and reinforcement learning steps. To evaluate whether the adaptive value generation and dynamic state merging techniques can affect the effectiveness and efficiency of ConfRL, we consider two “vanilla” versions of ConfRL. The first version ConfRL₁ does not apply dynamic state merging. The second version is ConfRL₀, which does not apply the adaptive value generation. Similar to MRND, ConfRL₀ assigns a random value to the configuration option. We let each technique run for 24 hours for as many iterations as it can complete before checking the results. To reduce the influence of randomness, we repeat each method for 10 times. The null-hypothesis, H₀ states that “the mean of the two methods are equal”, and we reject the null hypothesis if the probability value is less than 5% (p < 0.05). After all methods finish, we conduct the t-test to evaluate if the mean difference in each set of data is statistically significant.

7.5.3.2 RQ1: Effectiveness of ConfRL

RQ1 evaluates whether ConfRL is effective at guiding the applications toward higher performance by adjusting the values of configuration options. Since the first step of ConfRL is to identify performance-influencing configuration options to reduce the search space, we want to evaluate if the ranking is accurate compared to MRND. Specifically, ConfRL uses a 3-way covering array to conduct configuration sampling. The top-10 configuration options are returned, such configuration options get a higher weight in reinforcement learning. In other words, the action
issued from reinforcement learning is biased towards such options. \( M_{RN D} \), on the other hand, randomly selects 10 configuration options.

To measure the effectiveness of ranking, the mean average precision (MAP) score is used. MAP is a single-figure measure of ranked retrieval results independent of the size of the top list [121]. Next, we evaluate if ConfRL’s performance tuning algorithm is effective. To build the Q-Table, we need to obtain performance measurements. Benchmark tools are used to generate workload and measure performance. For instance, the Apache Benchmark (ab) is used: “ab -n 1000 -c 10 http://localhost”. -n specifies the number of requests and -c specifies the level of concurrency. The benchmark tools provide an elegant solution to generate synthetic traffic at demand.

In the experiment environment, all non-system processes are terminated to dedicate the system resource to the subject software and to reduce any other activities that may disturb the experiment. We configure each subject according to the performance tuning guidance [11, 80, 102, 133] and benchmark 1000 times for each subject to record the subject’s performance. The best performance is selected and used as the performance goal \( P_{GOAL} \). This performance is established as the maximum performance achievable in the experiment environment. We look at the mean performance achieved when each method reaches the time limit.

7.5.3.3 RQ2: Efficiency of ConfRL

RQ2 evaluates how long it takes for ConfRL to achieve a given performance measurement. We compare ConfRL with the three baseline techniques (i.e., \( M_{RN D} \), ConfRLA, and ConfRLD) to evaluate the overall efficiency of ConfRL. The mean performance measurement is calculated for every hour. We consider the reinforcement learning procedure converges when the mean performance measurement is within a 10% range of \( P_{GOAL} \) and maintains the same level of performance to the end of the experiment. Ideally, the mean performance measurement should be equal to \( P_{GOAL} \), due to the internal implementation of the benchmarking tool, the uncertainties on the experiment environment, and the nature of reinforcement learning method, it is not always possible to achieve a mean performance measurement equal to \( P_{GOAL} \). Although measuring performance is the most time-consuming operation, each method uses the same method (benchmark tool). Hence, the time spent on such steps is comparable.
7.6 Results and Analysis

7.6.1 RQ1 Effectiveness of ConfRL

Table 7.5 shows the result of the effectiveness of ranking performance-influential configuration options. The “MAP” column lists the mean average precision scores for both ConfRL and $M_{\text{N/D}}$. The MAP score of ConfRL ranges from 0.36 to 0.7, with an average MAP score of 0.52. ConfRL outperforms $M_{\text{N/D}}$ in three out of four cases. ConfRL successfully identifies at least one performance-influential option and ranks options in the top-10 position. The results show that the option ranking method used in ConfRL is effective.

In the “Effectiveness” column, we report the mean performance measurements across all the iterations for ConfRL, $M_{\text{N/D}}$, ConfRL_A, and ConfRL_D respectively. As the results suggest, ConfRL outperforms $M_{\text{N/D}}$ in all four programs, ranging from 14% to 30%, with 24% on average. The t-test shows the difference between two sets of data is statistically significant. The result shows that ConfRL can effectively select the right configuration options to optimize performance.

Figure 7.3 shows the plotting of four methods in each subject program. The plot shows how each method performs within the time limit. The x-axis indicates the timeline. The y-axis indicates the performance measurement. Due to the space limitation, we calculate the average performance measurement for each hour, hence for we have 24 data points in each plot. Each data point corresponds to the average performance measurement within that one hour time period.

Since there is only a small subset of states that can lead to higher performance, initially, all four methods seem to go hand in hand in terms of the average performance measurement. As a matter of fact, the $M_{\text{N/D}}$ can often achieve similar and sometimes better performance. In early iterations, the performance fluctuations in ConfRL, ConfRL_A, and ConfRL_D are expected. This is due to changes in the application states (i.e. states are represented by different combinations of configuration option values as illustrated in Section 7.4.6). In the early stage, the agent needs to explore as many states as possible to understand the environment. The fluctuations are also caused by epsilon-greedy exploration. Specifically, in order to explore more states, the agent does not follow exactly the best path learned from previous iterations.

There is a small chance (determined by the epsilon) that the agent may stray away from the current policy. The action is selected by following the $\epsilon$-Greedy algorithm. In a nutshell, $\epsilon$ determines the randomness of exploring outside of the learning comfort zone, this allows the agent to have a chance to explore unseen states. The agent either receives a random action in the exploration phase or an action choice by exploiting the past experience. This prevents the
agent from trapping at a local maximum. ConfRL gradually reduces $\epsilon$ to help the learning process converge faster. Towards the end of the execution, the performance measurement tends to stabilize as the agent figures out what actions to take for a given state. Because ConfRL uses fewer states in the learning process, we observe that ConfRL converges faster than ConfRL$_A$ and ConfRL$_D$. On the contrary, MRND does not learn from any previous interactions, the performance of the random method does not have noticeable improvement.

![Reinforcement Learning Plots](image)

Figure 7.3 Reinforcement Learning Plots

### 7.6.2 RQ2 Efficiency of ConfRL

The “Efficiency” column in Table 7.5 shows the efficiency of ConfRL. When comparing ConfRL to the baseline random method MRND, in three out of four subject programs, ConfRL uses less time to converge to the target performance. Once ConfRL converges, it takes minutes to guide the subject to the state that outputs the target performance. On average, ConfRL uses 20.5 hours to converge whereas all other methods fail to converge within the 24-hours’ time limit except one occasion in Lighttpd with ConfRL$_D$. Lighttpd has the smallest number of configuration options in all four subjects. The size of the configuration space hence the number of runtime states is smaller compared to other subjects. We conjecture it is the smaller size of
the runtime states that leads to the ConfRL method to converge faster. Nonetheless, the result shows that the dynamic state reduction technique is useful as the ConfRL method takes longer to converge.

Table 7.5 Effectiveness and Efficiency of ConfRL

<table>
<thead>
<tr>
<th>App.</th>
<th>MAP</th>
<th>Effectiveness</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfRL</td>
<td>MND</td>
<td>ConfRL</td>
<td>MND</td>
</tr>
<tr>
<td>Apache</td>
<td>0.59</td>
<td>0.17</td>
<td>4607 r/s</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>0.7</td>
<td>0.17</td>
<td>3864 r/s</td>
</tr>
<tr>
<td>MySQL</td>
<td>0.42</td>
<td>0.64</td>
<td>324 t/s</td>
</tr>
<tr>
<td>PSQL</td>
<td>0.36</td>
<td>0.11</td>
<td>248 t/s</td>
</tr>
</tbody>
</table>

Table 7.6 shows the results of ConfRL, ConfRL_A, and ConfRL_D for evaluating the impact of state reduction techniques. The result in Table 7.6 shows the number of states reduced ranges from 11% to 36% and on average ConfRL reduces reinforcement learning states by 22.8%. When comparing to the ConfRL_D method, the states reduced range from 10.7% to 79.8% and on average ConfRL uses 22.8% fewer states. ConfRL uses 26.3% to 82.5% (on average 62.3%) fewer states when compared to the ConfRL_A method. As we can see, ConfRL reduces the number of reinforcement learning states without losing learning power. The results show that the reinforcement learning state reduction techniques used in ConfRL are efficient.

Table 7.6 Impact of State Reduction Techniques

<table>
<thead>
<tr>
<th>Application</th>
<th>ConfRL</th>
<th>ConfRL_A</th>
<th>ConfRL_D</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>PM</td>
<td>States</td>
<td>PM</td>
</tr>
<tr>
<td>Apache</td>
<td>21696</td>
<td>4607 r/s</td>
<td>54846</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>15370</td>
<td>3864 r/s</td>
<td>20854</td>
</tr>
<tr>
<td>MySQL</td>
<td>19045</td>
<td>324 t/s</td>
<td>99956</td>
</tr>
<tr>
<td>PSQL</td>
<td>23587</td>
<td>248 t/s</td>
<td>134524</td>
</tr>
</tbody>
</table>

7.7 Discussion

7.7.1 Sensitivity of Strength

We evaluate ConfRL with a different initial state. Specifically, we conduct an experiment with three sets of initial states: 1) the options with default out of the box values; 2) the options with random values; 3) the options with known best performance values. In the first two cases, after some iterations (e.g. 200 iterations), the agent can guide the subject system toward better performance gains. However, in the third case (subjects configured with the known options for performance), the agent shows a zigzag pattern. The agent first goes to a state that leads to poor performance, and then comes back to a state that results in a good performance. It makes sense to the agent, as this would allow the agent to get more rewards in each iteration.
Experiments have shown that by giving a greater penalty to the agent may alleviate this phenomenon. This shows a potential weakness in ConfRL as it may not behave optimally when starting in the optimal state.

7.7.2 Threats to Validity

The primary threat to the external validity of this study involves the representativeness of our subjects. Other subjects may exhibit different behaviors. We reduce this threat to some extent by using several varieties of well-studied open-source projects from different application domains. For the same application type (e.g. a web server), we choose two subjects. These systems have varying numbers of configuration options.

The primary threats to the internal validity of this study are possible faults in the implementation of our approach and in the tools that we use to perform the evaluation. We control this threat by extensively testing our tools and verifying their results against a smaller program for which we can manually determine the correct results. For each test subject, we start with a small set of manageable configurations to test things out before conducting experiments on a larger scale. The time complexity of dynamic stage merging is proportional to the number of runtime states, it could be very expensive when the subject has an extensive number of runtime states. We control this threat by identifying the performance-influential configuration options and restricting the number of configuration option values to reduce the number of runtime states.

7.7.3 Limitations

First, this work does not evaluate the impact of multi-layer software systems. Instead, ConfRL treats the other layers in a black-box fashion. For instance, when requesting a web page from the Apache web server, the dynamic page could make calls to render the dynamic content on the web page from a backend database server. ConfRL does not consider the impact of the backend database server when adjusting the configuration options on the Apache server. However, in our setup, we make sure other layers in a multi-layer software have exactly the same setup throughout the experiments. Second, this work does not evaluate the impact of a resource sharing server where multiple software can request hardware resources at the same time. The current setup assumes that the subject systems are the only resource demanding applications on the host machine. We see this as a reasonable assumption as in practice many businesses would prefer to deploy web servers and database servers to dedicated machines.
### 7.7.4 Related Work

**Configuration Auto Fix.** Su et al. [135] proposed a causality dependency tracking and analysis approach on modified Linux kernels to help users to find a solution of the configuration problem. Swanson et al. [137] designed the REFRACT, a self-adaptive framework to find workarounds to fix and prevent future configuration-induced software failures. Whitaker et al. [150] proposed Chronus, a tool that utilized user provided probes to search through the incremental system checkpoints to find the offending states and diagnose configuration errors that caused software functional problems. Unlike our approach that targeted on the software performance (non-functional requirement) long-term gain, the aforementioned methods targeted on software failures (functional requirement) which are vastly different than performance issues. Several techniques have been proposed to find optimal configurations using machine learning techniques [41, 83]. Diao et al. [41] proposed an approach to use the fuzzy controller to automatically tune configuration options that were known to have a concave upward effect to optimize response time. One big limitation with this approach is that the method relied on the qualitative knowledge of selecting such configuration options. Liu et al. [83] conducted experiments to find the best optimization techniques to reduce response time by adopting online optimization methods, such as Newton’s Method, to configuration options in the Apache web server. However, Newton’s method based hill climbing techniques can be used to find the optimal value only when the problem has a concave upward effect on the parameter, therefore limiting its adaptability. Reinforcement learning, on the other hand, is known to solve the problem of determining what actions to take without requiring any prior knowledge of the environment. Naturally, it is suitable for the self-adaptive system problem.

**Reinforcement Learning Techniques.** Other literature [25, 113] explored the use of reinforcement learning in the context of dynamically adjusting resource allocations (e.g. CPU and Memory) on the resource sharing virtual machine environment. Such efforts were mainly focused on optimizing the hardware-level resource configurations on the virtual machine environment where guest systems may compete for shared resources. In such cases, the size of the configuration space was relatively smaller since only a handful of resources were needed to be considered. Also, the best practice for tuning performance on the hardware level is well established compared to the software level configurations. Bu et al. [25] proposed RAC, a reinforcement learning approach to automatically update the application configuration in response to the web traffic and virtual machine changes. Rao et al. [113] proposed a reinforcement learning approach to automatically configure resources on virtual machines (VM). In their work, the configuration space was defined in terms of the system resource allocations in the VM environment. The number of configuration options (CPU, MEM) to change was small. The configuration space is much bigger in our subjects, for instance, Apache has hundreds of configuration options. Also, the prior work focused on managing resources on the VM-level to
maximize throughput whereas we focused on achieving long-term performance gains on the application-level given fixed hardware resources.

**Control Theory Techniques.** Previous literature [5, 98, 147, 166] used of the control theory to manipulate configurations. The control theory works particularly well when certain constraints must not be violated. However, the use of the control theory will require extensive knowledge of the underlying system and a lot of effort in the hyper-parameters tuning. As previous performance bug empirical studies [147, 166] showed, application-level configurations may have a great impact on the overall application performance. Wang et al. [147] designed SmartConf to use the control theory to build a prediction model for each option to maximize software performance while maintaining the required operating constraints. SmartConf required code modification whereas our method does not rely on the source code to work. Zhang et al. [166] applied convergent control rules to design a framework that enabled friendly virtual machines which can adjust their demands based on feedback on the hardware resource usage and availability. Because of the differences in the project goals, authors of SmartConf agreed that machine learning based techniques are “better than controllers in deciding optimal settings.” Abdelzaher et al. [5] showed a feedback control theory to achieve response-time and throughput guarantees to different classes of clients in a general web server. Padala et al. [98] used a classical control theory to allocate resources dynamically to meet the application-level quality of service in a virtual data center environment. Our work, on the other hand, used reinforcement learning to train the agent to automatically adjusting configuration options to get long-term performance gains.

7.8 Conclusions

Performance is crucial to the success of software systems. While modern software often offers great flexibility through configuration files, the large number of configuration options can be difficult to understand and even more intimidating to setup properly. Previous studies have shown that the configuration options can have a great influence on software performance. Finding the right set of configuration options to improve performance is not a trivial task even for experts. It is desirable to have a mechanism that can automatically adjust configuration options in order to achieve higher performance gains.

In this paper, we present ConfRL, a reinforcement learning approach that automatically tunes performance-influential configuration options to achieve long-term performance gains. We evaluate ConfRL on four large-scale server projects. Our evaluation shows that ConfRL can efficiently achieve higher long-term performance gains up to 30%. Our experiment shows that ConfRL can effectively reduce the number of reinforcement learning states by as much as 82.5%. On average, ConfRL converges in 20.5 hours. In the future, we plan to study additional factors
that may influence the effectiveness and efficiency of ConfRL, such as the context of the system environment. We also plan to study if ConfRL can be used to correct performance bugs caused by misconfiguration.
In Chapter 3, we have conducted a performance bug characteristics study in highly-configurable systems. We have studied 300 configuration-related performance bugs from three major open source projects. We have examined a wide spectrum of performance characteristics in the context of modern highly-configurable software. This includes the prevalence of configuration induced performance bugs, the type of configuration options that could influence performance, the root cause of configuration related performance bugs, and the complexity involved to fix performance bugs. In the discussion session, we provide insights for both researchers and practitioners to benefit their work.

In Chapter 4, we shared 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 bug reports. We provided a set of workarounds to increase the chance of success in performance bug reproduction. We studied two large-scale open-source server projects. We randomly selected, analyzed, and conducted the reproduction of 93 bugs in total. Our study targeted at reproducing performance bugs from the perspectives of researchers. The study aimed to help researchers better understand the challenges in performance bug reproduction and propose solutions to facilitate the bug selection process. We plan to extend the study by replicating non-performance bugs and performance bugs that have not been fixed or confirmed. We want to compare and contract for the experience and findings in other types of bugs to find out if the study findings in this work are exclusive to the fixed performance bugs or maybe it is overlapping with replicating other types of bugs.

In Chapter 5, we have presented a configuration-aware performance profiling approach. The profiling phase utilizes the dynamic instrumentation technique to actively monitor and track conditional instructions in loop structures. Such profiling data is further processed to fit each identified loop into a set of predefined machine learning fitting functions. The performance score is then calculated to provide a ranked list of configuration options. We plan to extend this work by incorporate other types of code regions that may also lead to performance problems. For instance, we want to include code locations that may lead to thread contentions. It would also be interesting to consider how environmental factors may influence the software performance.

In Chapter 6, we manually identified and analyzed 300 performance bug reports from three popular open source projects. We discovered that it might be possible to leverage information retrieval and natural language processing techniques to extract commands and input parameters from bug reports. We found that some input parameters were more likely to cause performance bugs and should be used with higher priority in performance testing. Based on our findings, we developed PerfLearner, an approach that combines natural language processing
and information retrieval to automatically extract relevant commands and input parameters from bug reports and use them to generate performance test frames for guiding performance testing. For future work, we plan to extend the work by utilizing association rules to automatically generate the test frame element constraints. We are also interested to find out if we may be able to extract frame elements from other sources such as Stack Overflow to generate high-quality test frames.

In Chapter 7, we proposed an automatic performance tuning approach, ConfRL, that can automatically select and tune configurations in response to environmental dynamics to optimize the system’s performance. The key idea of ConfRL is to use reinforcement learning that enables an agent to learn in the runtime environment by trial-and-error and use the feedback to tune configuration options to achieve higher performance. Our evaluation of four real-world highly-configurable open-source server projects showed that ConfRL can efficiently and effectively guide software systems to achieve higher long-term performance gains. We plan to evaluate the efficiency and effectiveness of ConfRL on correcting misconfiguration bugs out of the faulty state with the current implementation. We also want to conduct a thorough study on what the best timing is for the reinforcement learning to initiate an action for performance adjustment on the production environment.


[127] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller,


Education

Master of Science, Old Dominion University, Computer Science

Bachelor of Management, North University of China, Information System

Professional Positions

Teaching assistant, University of Kentucky, 2014

Research assistant, University of Kentucky, 2016

Developer, Liberty Tax Service, 2008

Publications

Xue Han, Tingting Yu, David Lo, PerfLearner: Learning from Bug Reports to Understand and Generate Performance Test Frames, IEEE/ACM International Conference on Automated Software Engineering (ASE), 2018


Yu, Tingting and Wen, Wei and Han, Xue and Hayes, Jane Huffman, ConPredictor: Concurrency Defect Prediction in Real-World Applications, Transactions on Software Engineering (TSE), 2018

Jane Huffman Hayes, Wenbin Li, Tingting Yu, Xue Han, Mark Hays, Clinton Woodson, Measuring Requirement Quality to Predict Testability, 2nd International Workshop on Artificial Intelligence for Requirements Engineering (AIRe), 2015