DEAN, DANIEL JOSEPH. Unsupervised Performance Anomaly Management for Production Cloud Environments. (Under the direction of Dr. Xiaohui (Helen) Gu.)

Ensuring satisfactory application performance in multi-tenant cloud environments is a challenging task. Despite extensive testing, many performance problems are missed during development and carried over into production. These problems can come from a variety of sources such as environment factors, interference from other co-located applications and software bugs. When a problem manifests in the production environment, there can be major financial penalties for all parties involved.

Due to the complex and distributed nature of production systems, these performance problems are inevitable. Instead of reacting to a production-run problem when it occurs, we instead propose a framework designed to proactively manage these issues.

First, we present UBL, a tool for black-box performance anomaly prediction in cloud environments. UBL is able to predict when a performance anomaly will occur by monitoring black-box metrics such as CPU usage, memory usage, and network usage using an unsupervised artificial neural network called the self organizing map (SOM). Our results show that UBL is able to predict anomalies with higher accuracy than other alternative approaches such as principle component analysis. Additionally, UBL is lightweight, imparting negligible overhead to the tested systems.

Second, we present PerfCompass, a tool for online fine-grained fault localization using system call traces. When a system experiences a performance problem, as predicted by UBL or a similar tool, PerfCompass is triggered to trace the system calls for the failing application. We use this trace to first localize the problem as either an internal software bug or external environmental cause using a novel two-phase differentiation scheme. The first phase looks at the percentage of threads experiencing a significant increase in execution time or frequency while the second phase considers how long it takes each thread to be affected by the problem. Our results indicate we are able to correctly localize all 24 problems we tested while imparting an average of 2.1% runtime overhead to the server.

Third, we describe PerfScope, a fine-grained root cause inference tool. When a problem is localized to be an internal software bug, PerfScope provides developers with a ranked list of suspicious functions for inspection. PerfScope uses a combination of clustering along with an unsupervised data mining technique called frequent episode mining on large system call traces. Our results show that PerfScope is effective, providing developers with a short list of candidate cause related functions to examine in 12 real software bugs while imparting an average of 1.8% runtime overhead to the tested server applications.
Finally, we present HSR, a hybrid static-runtime analysis tool. HSR combines rule-based static analysis with runtime diagnosis hints (i.e., cause related functions identified by PerfScope) in order to reduce the number of non-bug related functions (i.e., false positives) developers need to examine. Our results indicate HSR is able to reduce the number of false positives by up to 98% compared to static approaches and up to 91% compared to runtime approaches while also covering root-cause related functions.
Unsupervised Performance Anomaly Management for Production Cloud Environments

by

Daniel Joseph Dean

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2015

APPROVED BY:

Dr. William Enck

Dr. Frank Mueller

Dr. Min Chi

Dr. Xiaohui (Helen) Gu
Chair of Advisory Committee
DEDICATION

To my family and friends. I couldn’t have done this without your love and support.
Daniel J. Dean grew up in Riverhead NY, a small town on the east end of Long Island. He received a BS and MS in computer science from Stony Brook University, New York in 2007 and 2009 respectively. After working in Charleston, SC for a brief stint, he joined the PhD program at North Carolina State University where he has been working under the guidance of Dr. Xiaohui (Helen) Gu since May of 2011. His general research interests are in computing systems with a focus on production system performance anomaly management, distributed systems, and cloud environments. His PhD dissertation research involves developing a framework to help predict and diagnose performance anomalies in cloud environments. He has interned with NEC Labs America in the summer of 2012 and IBM Research in the summer of 2013. Daniel is a student member of the IEEE.
ACKNOWLEDGEMENTS

I would not have been able to tackle the task of developing this dissertation without the encouragement, feedback, and support of several people, which I would like to acknowledge now.

First and foremost, I would like to thank my advisor Dr. Xiaohui (Helen) Gu. Her drive, passion about her work, and knowledge of cloud systems inspired me to work much harder than I ever thought possible over the past several years. In particular, working with her has taught me how important attention to detail is to success in both research and in life; a valuable lesson learned. I truly enjoyed working with her and it is an experience I will fondly remember.

I would next like to thank my dissertation committee members, Dr. Bojko Bakalov, Dr. Min Chi, Dr. William Enck, and Dr. Frank Mueller, for their insight and feedback on my dissertation.

I would like to thank Dr. William Enck for making time to talk with me about research and for his feedback regarding my faculty application.

I want to thank Dr. Guoliang Jin for the discussions we have had about static analysis and for pointing me in the right direction when deciding which static analysis approaches to implement.

I would also like to thank several current and former labmates, Dr. Hiep Nguyen, Peipei Wang, Anwesha Das, Vivek Nair, Anson Ho, Kamal Kc, and Chin-Jung Hsu for their valuable friendship and feedback. In particular, I would like to thank Dr. Hiep Nguyen for his encouragement when graduation seemed a lifetime away.

I would like to thank the National Science Foundation and the United States Army Research Office for the funding they have provided which made my research possible.

Lastly, I would like to thank all my family and friends for their love and support. This would have not be possible without you guys.
# TABLE OF CONTENTS

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

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

**Chapter 1 Introduction** ......................................................... 1
  1.1 Motivation ................................................................. 2
  1.2 Summary of the State of the Art ......................................... 3
  1.3 Thesis Statement ............................................................ 4
    1.3.1 Research Challenges .................................................. 4
  1.4 Summary of Contributions .................................................. 5

**Chapter 2 System Overview** ................................................... 7
  2.1 UBL ................................................................. 7
  2.2 PerfCompass ............................................................... 8
  2.3 PerfScope ................................................................. 9
  2.4 HSR ................................................................. 10

**Chapter 3 UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems** ............................................. 11
  3.1 Introduction ............................................................... 11
  3.2 System Design ............................................................. 12
    3.2.1 Online System Behavior Learning ..................................... 13
    3.2.2 Unsupervised Anomaly Prediction ..................................... 16
    3.2.3 Anomaly Cause Inference ............................................. 19
  3.3 Experimental Evaluation .................................................. 20
    3.3.1 Evaluation Methodology ............................................... 20
    3.3.2 Results and Analysis ................................................ 23
  3.4 Summary ................................................................. 28

**Chapter 4 Online Faulty Function Localization for Debugging Performance Anomalies in Cloud Hosting Infrastructures** ............................................. 29
  4.1 Introduction ............................................................... 29
  4.2 Preliminaries ............................................................. 32
    4.2.1 System Model .......................................................... 32
    4.2.2 Problem Formulation and Key Assumptions ............................ 32
  4.3 System Design ............................................................. 33
    4.3.1 System Overview ...................................................... 34
    4.3.2 Execution Unit Extraction ............................................ 35
    4.3.3 Fault Onset Time Identification ..................................... 36
    4.3.4 Fault Differentiation Schemes ....................................... 38
    4.3.5 Affected System Call Ranking ....................................... 38
    4.3.6 Fault Localization Enhancement ..................................... 39
  4.4 Evaluation ................................................................. 41
Chapter 5 PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures

5.1 Introduction ............................................. 58
5.2 System Design and Implementation .......................... 62
  5.2.1 Offline Function Signature Extraction ..................... 62
  5.2.2 Online Bug Inference .................................. 66
5.3 Experimental Evaluation .................................... 69
  5.3.1 Real Performance Bug Samples ............................ 73
  5.3.2 Experiment Setup ...................................... 73
  5.3.3 Bug Inference Result Summary ......................... 74
  5.3.4 Bug Inference Case Studies ......................... 75
  5.3.5 Sensitivity Study ..................................... 78
  5.3.6 PerfScope Overhead ................................... 78
5.4 Summary ............................................. 80

Chapter 6 HSR: Hybrid Static-Runtime Analysis for Performance Anomaly Diagnosis

6.1 Introduction ............................................. 81
6.2 Design .................................................. 84
  6.2.1 System Overview ...................................... 84
  6.2.2 Rule-Based Static Analysis ......................... 85
  6.2.3 Hybrid Function Identification ..................... 87
6.3 Evaluation ............................................. 91
  6.3.1 Implementation ...................................... 91
  6.3.2 Experiment Methodology ............................ 92
  6.3.3 Rules Matched ...................................... 94
  6.3.4 Ranking Results ................................... 95
  6.3.5 Reduction Results ................................. 97
6.4 Summary ............................................. 98

Chapter 7 Related Work ........................................ 101
7.1 Performance Anomaly Prediction .......................... 101
7.2 Performance Anomaly Localization ..................... 102
7.3 Performance Bug Diagnosis ............................ 104
7.4 Static Analysis Techniques ........................... 106
7.5 Dynamic Analysis Techniques .......................... 107
7.6 Hybrid Analysis Techniques ........................... 107

Chapter 8 Conclusions and Future Work ...................... 108
8.1 Conclusions ........................................... 108
8.2 Future Directions .......................................................... 109

References ........................................................................... 112
LIST OF TABLES

Table 3.1 UBL System overhead measurements. ........................................ 27
Table 3.2 Sensitivity experiment results for the NetHog fault in RUBiS and MemLeak fault in System S. ................................................ 27
Table 4.1 Notations. ........................................................................... 33
Table 4.2 Descriptions of the 17 external faults we tested. ................. 51
Table 4.3 Descriptions of the 7 internal faults we tested. ................. 52
Table 4.4 System call ranking results for the external faults we tested for C/C++ systems. .............................................. 53
Table 4.5 System call ranking results for the external faults we tested for Java systems. .............................................. 54
Table 4.6 System call ranking results for the internal faults we tested. .............................................. 55
Table 4.7 The fault impact factor ($\tau$) results under different parameter settings of the PerfCompass system. The numbers in bold represent the default settings we used. .............................................. 56
Table 4.8 PerfCompass analysis time for each fault we tested. ....................... 57
Table 5.1 Statistics on the real performance bugs we examined. .......... 69
Table 5.2 Reproduced real C/C++ performance bugs that are used in our experiments. .............................................. 70
Table 5.3 Reproduced real Java performance bugs that are used in our experiments. .............................................. 71
Table 5.4 Online bug inference result summary. ..................................... 72
Table 5.5 Online and offline computation time for PerfScope. ............... 79
Table 6.1 Descriptions for the static rules we tested. ............................... 93
Table 6.2 Descriptions of the real-world bugs tested. .............................. 93
Table 6.3 The total number of times each rule is matched across all bugs we tested. .............................................. 95
Table 6.4 Results of combining runtime and static analysis for diagnosis. .............................................. 95
Table 6.5 The bug related functions we identified using LLVM/Findbugs, PerfScope analysis, and HSR schemes. .............................................. 96
Table 6.6 The percent reduction in false positives achieved when combining runtime and static analysis for diagnosis. .............................................. 100
# LIST OF FIGURES

Figure 2.1 The overall design of our framework. When a performance anomaly is predicted by UBL for a monitored application (e.g., Hadoop), an alert is generated and a shadow copy of the production environment is created if necessary using existing tools. PerfCompass is then triggered to localize the anomaly with PerfScope being triggered to provide diagnostic information for internal software bugs. Finally, HSR is triggered to provide more accurate runtime hints if a static profile is present for the monitored application. .............. 8

Figure 3.1 SOM training process. ................................................. 13
Figure 3.2 An example path showing the system evolution from normal to failure. .... 17
Figure 3.3 Grey-scale visualization of the SOM models for the RUBiS with the Network Hog fault and System S with the MemLeak fault. Darker neurons have larger neighborhood area sizes while lighter neurons have smaller neighborhood area sizes. ............................... 18
Figure 3.4 Performance anomaly prediction accuracy comparison for RUBiS under different faults. ................................. 22
Figure 3.5 Performance anomaly prediction accuracy comparison for IBM System S under different faults. ..................... 22
Figure 3.6 Performance anomaly prediction accuracy comparison for Hadoop under different faults. .............................. 23
Figure 3.7 The achieved lead time by the UBL anomaly prediction model. ............. 24
Figure 3.8 Ranking results of the faulty metrics in different failure instances. The Y axis is the faulty metric rank as determined by UBL while the X axis represents the total number of faults observed. ............................... 26

Figure 4.1 PerfCompass overview. ............................................. 30
Figure 4.2 Fault onset time calculation for one execution unit. The moving average for each system call type is calculated separately. The moving average of `sys_write` increases from 20ms in window 1 to 19ms in window 2 to 60ms in window 3. The time elapsed from the start of the execution unit \( \lambda_1 (t_0) \) to the currently processed system call (e.g., `sys_write` in the current window \( t_7 \)) is defined as the fault onset time \( \sigma \). ......................... 36
Figure 4.3 PerfCompass external and internal fault localization algorithm. .......... 40
Figure 4.4 The computed fault impact factor \( \tau \) using calibrated fault onset time threshold \( \alpha \). ........................................ 43
Figure 4.5 The normalized fault onset time dispersion \( \Omega \) values using calibrated fault onset time threshold \( \alpha \). ...................... 44
Figure 4.6 The computed fault impact factor \( \tau \) using a 0.5 second static fault onset time threshold \( \alpha \). ......................... 45
Figure 4.7 The normalized fault onset time dispersion \( \Omega \) values using a 0.5 second static fault onset time threshold \( \alpha \). ...................... 46
Figure 4.8 The filtering mode effect using calibrated fault onset time thresholds for different faults where the filtering mode is triggered.

Figure 4.9 The overhead imposed by PerfCompass on each of the tested server systems.

Figure 5.1 A subset of function call graph for a real HDFS performance bug. When an input file larger than 2GB is used, the internal variable (int) representing the content length of the input file overflows. This causes the call to `in.read(buf)` at line 81 to never return -1 causing an infinite loop. The bug point is highlighted in bold.

Figure 5.2 System call trace excerpt from a Tomcat application server run.

Figure 5.3 Function signature profiling. By combining the function execution list and the system call trace via the active function stack, we label each system call with the function that produces it.

Figure 5.4 Frequent closed system call episodes discovery. All the frequent system call episodes (counter > 10) are highlighted in gray color. The frequent closed episode is highlighted within a circle.

Figure 5.5 A subset of the call graph for the Cassandra bug. The bug causes a clean memtable not to be added to the list of memtables, resulting in an infinite loop in the `reload` function. The bug point is highlighted in bold. PerfScope identifies the root cause function `maybeSwitchMemtable` as the 4th top ranked bug-related function, which is responsible for adding all the memtables.

Figure 5.6 A subset of the call graph for the Lighttpd bug. The errno is handled incorrectly, causing the main function to process the same event again and again. The bug point is highlighted in bold. PerfScope identified the `fdevent_poll` as the bug related function which is responsible for controlling the event processing to continue as long as the number of events (n) is positive.

Figure 6.1 The patch for Apache-37680. The bug occurs as a result of the constant value 1 being passed to the `apr_socket_opt_set` function, causing an endless loop in another function at runtime.

Figure 6.2 An overview of HSR. Our system takes bitcode or bytecode from a monitored application (e.g., Hadoop) as input and produces a possibly buggy function profile. We then combine that profile with runtime information to produce a ranked list of HSR buggy functions.

Figure 6.3 The patch for MySQL bug 28000. The bug occurs as a result of the constant value 0 being passed to the `fill_record_n_invoke_before_triggers` function, causing an endless loop at runtime.

Figure 6.4 Simple Java source code showing FunctionA calling FunctionB.

Figure 6.5 The result of decompiling the simple source code shown in Figure 6.4. Function call to FunctionB is transformed into an `invokedynamic` instruction with the parameters being the `iconst_1` instruction before it.

Figure 6.6 The message template for HSR. Items shown in brackets allow users to quickly develop new rules for HSR with few code changes.
Figure 6.7  The type of functions we can identify. The functions likely to cause a bug are in the intersection of the two sets shown. That is, those functions with buggy characteristics that have experienced a problem at runtime. 88
Figure 6.8  The call paths for an example application. The dotted lines represent possible call paths while the solid lines represent the actual runtime call path taken. 89
Figure 6.9  The function invocation message template for HSR. This template allows HSR to be easily extended to perform a variety of control flow analysis as needed. 90
Figure 6.10  The call stack for a traced application for thread tid = 5. When a function at time $T_{current}$ is pushed on to the stack we output a message indicating that the function at $T_{top}$ called function at $T_{current}$. We then increment $T_{current}$ to the next entry. 91
Figure 6.11  The number of false positives identified for each scheme for each Java-based bug we tested. As shown, combining runtime analysis with static analysis causes significantly fewer false positives to be identified while also identifying bug-related functions. 98
Figure 6.12  The number of false positives identified for each scheme for each C/C++-based bug we tested. As shown, combining runtime analysis with static analysis causes significantly fewer false positives to be identified while also identifying bug-related functions. 99
Chapter 1

Introduction

The idea of cloud computing has been around since the late 1990’s and the idea of time shari-
ing resources among multiple users to give the illusion of vast computing resources has been
around long before that. The term “the cloud” can refer to a variety of different items. In
an Infrastructure-as-a-Service (IaaS) cloud, infrastructure providers can provide access to bare
VMs in a pay as you manner [1, 7]. In a Platform-as-a-Service (PaaS) cloud, providers provide
a pre-configured ready version of an application or set of applications (e.g., Apache, Hadoop)
to users [55]. Both IaaS clouds and PaaS clouds have advantages and disadvantages. When
developing software for use in the cloud, also called Software-as-a-Service (SaaS), careful con-
sideration must be given when deciding whether to use an IaaS platform or PaaS platform.
For example, PaaS clouds usually hide many of the details of application configuration and
virtualization from the user. This, however, comes at the expense of limited flexibility (e.g.,
creating new virtual machines).

The virtualization technology that makes the cloud possible creates an abstract interface
to part or all of the underlying physical resources using a special piece of software called the
hypervisor. The guest operating systems run by the hypervisor, or virtual machines (VMs),
interact with the provided abstraction layer in order to gain access to physical resources (e.g.,
disk, network). Until recently, VMs were the finest level of virtualization granularity available.
Dockers [45], however, have made it possible to run virtualized applications by themselves with
less overhead than VMs.

In recent years, we have begun to see wide-scale adoption of the cloud for several reasons.
Among the top reasons [123] are: 1) elasticity; and 2) efficiency. First, the cloud platform can
offer elastic access to computing resources by taking advantage of the fact that users typically do
not use all the resources a physical machine can offer at any point in time. This allows providers
and users to quickly redistribute computing resources in the face of rapidly changing resource
demands. Second, the cloud allows software developers to be more efficient by allowing them to
concentrate on software development, not infrastructure management. Similarly, infrastructure providers can concentrate on providing highly reliable computing platforms as a service to users.

The majority of modern software used in the cloud runs in two vastly different environments. The first environment is the development environment. This environment is designed to give as much feedback as necessary to software developers to allow them to efficiently find and fix most of the bugs existing in the software being developed. In this environment, performance is not a major consideration. The second environment, the production environment, is designed to face users and is optimized for performance. Here much of the debugging feedback present in the development environment is disabled in order to provide better performance to end-users.

When a problem manifests at runtime as an unexpected application performance degradation, we call that a performance anomaly. For example, an increase in CPU usage causing an unexpected increase in response time for an Apache web-server would be considered a performance anomaly. When a performance anomaly occurs in the production environment, where performance is critical, both long lasting [11] and short term [89] outages can have severe financial implications.

Performance anomalies occurring in a production environment are typically more difficult to diagnose compared to other problems (e.g., application crash) because they leave few clues behind for developers to use to determine what happened. For example, an internal software bug causing an application to crash will likely leave behind clues (e.g., core dump, error messages). An internal software bug causing an application to hang due to an infinite loop (i.e., a performance anomaly) may not leave any clues for developers because normal code paths are executing in an infinite loop.

Due to the high impact these anomalies have when they occur at runtime in production along with the long time they take to diagnose, developing automated tools and techniques to manage (i.e., predict, localize, and diagnose) production-level runtime performance anomalies is paramount to the future success of cloud computing. Furthermore, in order for these tools to effectively mitigate the effects of performance anomalies in production-level cloud environments, they must be lightweight, scalable, and able to identify both known and unknown problems.

1.1 Motivation

Although the cloud holds great promise, the added complexity of virtualization coupled with the distributed nature of many applications running in the cloud means it is easy for problems to occur. In fact, previous work has identified that both external environment issues [135] and internal software bugs [70] are significant issues faced in the cloud. Furthermore, the complexity of the cloud will likely increase in the coming years as we see finer degrees of virtualization used (e.g., Docker [45]). This is because as monolithic software is broken into smaller containers
interacting with each other, the number of unexpected interactions can increase. For example, there may be a greater likelihood for one small container of many interacting containers to be misconfigured compared to the chance that a single monolithic application is misconfigured. This paves the way for more performance anomalies to occur.

Although developers spend a huge amount of resources performing rigorous pre-deployment testing, ensuring 100% testing coverage is a challenging task. In addition, even if developers can fully test the interactions of all the functions they have developed, it is difficult to anticipate or test the results of third-party software interactions. This means that many problems go unnoticed, making their way into deployed code running in production cloud environments. As a result, software defects occurring in production-level deployments are inevitable. Instead of reacting to these problems when they occur, tools which proactively predict, localize, and diagnose problems in production cloud environments are critical to the future success of cloud computing.

1.2 Summary of the State of the Art

A large body of research has been focused on predicting and detecting performance anomalies under different settings. Previous work can be broadly classified into supervised methods [28, 29, 32, 39, 52, 93, 110, 124] and unsupervised methods [35, 57, 77, 96, 99, 103, 104]. Supervised methods rely on using an expert to first label data which is then used to detect or predict previously known anomalies whereas unsupervised methods do not require any expert labelling. Supervised methods are very good at identifying previously known problems, but not very good at identifying new problems. In contrast, unsupervised methods such as the methods we use can identify both known and unknown problems.

Much work has been done to diagnose performance problems. These works can be classified into a variety of groupings such as system event trace-based (e.g., system call) approaches [26, 44, 49, 85, 98, 99], network trace based approaches [17, 25, 33, 34, 48, 54, 95–97, 128], or console log based approaches [127, 130–132]. The system event trace based approaches typically impart the lowest overhead, however they also can be harder for developers to understand as they involve low-level metrics. Network trace based approaches are very good for localizing and debugging network-based issues, but are difficult to use for other problems. Lastly, console log approaches work very well for problems producing console logs. However, it is difficult to use those approaches on systems with limited logging capabilities. Additionally, there exist a variety of record and replay techniques [22, 51, 67, 102, 112]. These techniques typically use lightweight methods to record data on the server and then use heavy analysis modules offline when replaying the recorded data. Ensuring accurate offline replay using these approaches without being intrusive is a challenging task.
Work has been done using static rule-based checking [23] to find problems at compile time. Similarly, static analysis has been used to find and fix concurrency bugs [72, 114, 115, 119, 121, 125] and failing executions [86, 126]. Dynamic analysis has also been used to fix concurrency bugs [66, 69, 94, 116, 134] and failing executions [24, 74, 100]. However, it is difficult to adapt the static detection rules used by these tools to detect the bugs we examine.

1.3 Thesis Statement

The aim of this dissertation is to understand the limitations of existing automated tools for managing runtime performance anomalies and to suggest new techniques for advancing the state of the art. We have found that a large amount of system-level data (e.g., system metrics, system call traces) can be inexpensively collected online at runtime and those system-level data can be used to help manage performance anomalies at runtime. The discoveries and findings made during our studies formulate the following thesis statement:

Unsupervised online learning algorithms using easy to collect metrics such as system-level metrics (e.g., CPU usage, network traffic) and system calls can enable automated performance anomaly prediction and diagnosis in production cloud environments.

Parts of this thesis statement are present in each of our works. Furthermore, we believe it is possible to build a framework based on these observations which can significantly reduce the costly downtime of production-run performance anomalies.

Although the techniques we have developed are applicable to a wide-variety of cloud and non-cloud-based applications, they are specifically designed to work well online in production-level cloud environments. Therefore, the applications we have studied are all commonly used in the cloud. Similarly, the environment issues and internal software bugs we have studied are those that commonly occur in the cloud. Lastly, as hang bugs are difficult to diagnose and have a serious impact when they occur, we focus a large portion of our work on those hang bugs.

1.3.1 Research Challenges

The development of a framework to efficiently manage production run performance anomalies involves overcoming the following research challenges:

- **Unexpected problems:** Production run performance anomalies are often the result of problems which have not been previously encountered in testing. As a result, we need a framework which is able to handle both previously known and unknown problems.
Online operation: Due to the severe financial cost performance anomalies cause, it is desirable to predict and prevent a problem before it can cause any downtime. If this is not possible, finding and fixing the root cause of the problem as quickly as possible is a top priority. As a result, any framework should be able to operate online, providing results in a matter of seconds or minutes as opposed to hours or days.

Black-box: Source code is not always available in the production environment. In addition, a wide variety of different types of applications are typically run in production-level cloud environments. Therefore, any framework developed should treat applications as a black-box, not requiring source code or any application specific information.

Low overhead: Code running in a production environment typically has strict performance requirements. This means that interacting with applications running in this environment must be done while imposing a bare-minimum of overhead to those applications.

1.4 Summary of Contributions

In this dissertation, we make the following contributions:

- We present UBL, a novel black-box unsupervised behavior learning and anomaly prediction system for IaaS clouds [40]. UBL uses Self-Organizing Maps (SOM) to capture emergent system behavior and predict unknown anomalies by monitoring black-box metrics such as CPU usage, memory usage, and network usage. UBL is able to predict a range of performance anomalies with 5.9-87.7% higher true positive rates and 3.3-84.5% lower false alarm rates than similar alternative schemes. UBL is also able to achieve sufficient lead time in most cases for the system to take preventative action to prevent a failure. Lastly, UBL is lightweight imposing negligible overhead to the system it is monitoring.

- We present PerfCompass, a runtime performance anomaly fault localization tool using online system call trace analysis techniques. PerfCompass can identify whether a production-run performance anomaly is caused by an external fault (e.g., interference from other co-located applications) or an internal fault (e.g., software bug). PerfCompass does not require any application source code or runtime instrumentation, making it practical for use in production IaaS clouds. We evaluate PerfCompass using a set of popular software systems (e.g., Apache, MySQL, Squid, Hadoop, Cassandra) along with a variety of common cloud environment issues and real software bugs. Our results show PerfCompass is able to diagnose all 24 faults we tested while imparting an average of 2.1% of runtime overhead.
- We present PerfScope, a *practical* online performance bug inference tool to help the developer understand how a production-run performance anomaly caused by an internal software bug happened. PerfScope does not require source code or expensive online instrumentation, making it practical to use in a production system. Additionally, PerfScope operates *online*, producing a rank list of bug related functions right after a production-run performance anomaly occurs without requiring record and replay. We have implemented PerfScope and tested it by running a range of commonly used open source server systems (Apache, lighttpd, MySQL, Hadoop, Cassandra, Tomcat). Our experiments used a set of *real* performance bugs that are randomly sampled from the open source system bug repository. Our results show that PerfScope can narrow down the root cause functions to a small percentage (0.001-2.3%) of functions and is able to successfully identify the root cause related functions. Lastly, PerfScope imposes negligible runtime overhead (0.8-3.3%) to the production system during normal execution time.

- We present HSR, a fine-grained bug diagnosis tool which uses runtime information along with static analysis for performance anomaly diagnosis. Experiments we have conducted on real performance bugs indicate that combining runtime information with static analysis can significantly reduce the number of false positives reported compared to other approaches. Specifically, HSR is able to reduce the number of false positives by up to 98% compared to static approaches and up to 91% compared to runtime approaches.

This dissertation is organized as follows. Chapter 2 gives an overview of our framework. Chapter 3 describes the first component in our anomaly management framework, a black-box unsupervised anomaly prediction system named UBL. In Chapter 4, we present the second component in our framework, a fine-grained fault localization tool called PerfCompass. Chapter 5 describes the third component in our anomaly management framework, PerfScope a fine grained faulty function identification tool. Chapter 6 discusses HSR, our work with combining static analysis and runtime information to improve performance anomaly diagnosis. Chapter 7 describes the work related to our work. Finally, Chapter 8 concludes our discussion of performance anomaly management and describes future directions for our work.
Chapter 2

System Overview

This chapter provides an overview of our performance anomaly management framework along with a brief discussion of the research problems each component is designed to address. We have designed this framework to work well in production-level cloud environments. However, the approaches used are not tied to the cloud and could be applied to both virtualized and non-virtualized environments.

Figure 2.1 shows the overall design of our framework. As shown, our framework consists of four main components. We now discuss each component in detail.

2.1 UBL

Predicting production run performance anomalies is a challenging task because they often involve multiple metrics and they are frequently the result of a new problem. This makes adapting existing models a difficult task. Some approaches (e.g., linear regression) are unable to effectively handle multiple metrics while other approaches (e.g., signature-driven) can only identify previously known problems.

Our first contribution is UBL, a tool capable of the unsupervised prediction of known and unknown performance anomalies. The tool we developed to address this challenge was UBL. At its core, UBL uses an unsupervised artificial neural network called the Self-Organizing-Map (SOM) to learn the patterns of black-box metrics (e.g., CPU usage, memory usage). We choose the SOM for this task because of its scalability with large datasets along with its ability to effectively model data consisting of multiple metrics. We only consider black-box metrics because they are easily accessible in production cloud computing environments. UBL learns the normal operating patterns of an application and then looks for any changes from those normal patterns in order to predict both previously known and unknown performance anomalies. Lastly, UBL is able to give coarse-grained hints about what metrics are contributing
most to a performance anomaly (e.g., CPU, memory). The lead time and hints provided by

![Diagram](image)

Figure 2.1: The overall design of our framework. When a performance anomaly is predicted by UBL for a monitored application (e.g., Hadoop), an alert is generated and a shadow copy of the production environment is created if necessary using existing tools. PerfCompass is then triggered to localize the anomaly with PerfScope being triggered to provide diagnostic information for internal software bugs. Finally, HSR is triggered to provide more accurate runtime hints if a static profile is present for the monitored application.

UBL are designed to be used by system administrators or automated tools to take action to prevent a failure from occurring (e.g., service-level objective violation). These actions can be as simple as allocating more memory to a virtual machine (VM) which is experiencing a larger workload than usual. However, sometimes the fix is not clear or simple. In these cases, the alert generated by UBL is used to trigger the second component in our framework, PerfCompass.

### 2.2 PerfCompass

The source of a performance anomaly is not always clear in the cloud. Performance anomalies can be caused by internal software bugs which occur at runtime as a result of unexpected inter-
actions. For example, an unexpected return value can trigger an endless loop in an application. Performance anomalies can also be caused by the cloud itself. For example, a co-located virtual machine performing an I/O intensive workload can cause other virtual machines on the same physical host to experience a delay when performing any I/O operations.

Differentiating between these two types of problems is critical as the steps taken to debug them are very different. However, differentiating between external environment issues and internal software bugs is a challenging issue in the cloud. Although there are existing approaches to fault localization, they are difficult to adapt to this problem. For example, existing approaches can identify a faulty component in a distributed system, but only at a coarse grained (e.g., VM) level.

Our second contribution is PerfCompass, a tool designed to provide additional localization and diagnosis information. Specifically, PerfCompass is triggered by a production-run performance anomaly prediction tool (e.g., UBL) to perform fine-grained fault localization. To avoid interfering with the production environment, we can first leverage existing tools [82] to create a shadow copy of the production environment. PerfCompass then begins monitoring system calls in a fixed-size kernel buffer using a lightweight system call tracing tool [43]. PerfCompass next analyzes these system calls in order to quantify the impact of the performance anomaly on the system as globally direct or locally direct. Finally, PerfCompass uses this information to suggest the cause of the performance anomaly as either an external cloud-based issue or internal software bug while also outputting the system calls most affected by the performance anomaly.

When a problem is localized to an external cloud-based issue, simple steps can be taken based on the hints provided by PerfCompass to attempt to fix the problem (e.g., live migrate to a new host). However, for internal software bugs, developers need to diagnose and fix the application code. When an internal software bug is identified we trigger the third component in our framework, PerfScope, to provide additional feedback to help developers quickly fix bug.

2.3 PerfScope

When a problem occurs in the production environment, developers need to first reproduce the problem in order to debug it. However, the production environment is typically very different from the development environment. Code running in the production environment is typically optimized for speed while code running in the development environment provides as much feedback as possible. As a result of this, when a problem occurs in the production environment, reproducing the bug in the development environment is a challenging task. Debugging output (e.g., logging) is typically done at reduced rate in the production environment in order to improve application performance. This means messages critical for debugging may not be present
in production-level runtime logs. In addition, the environment itself (e.g., system libraries) may be significantly different in the production environment compared to the debugging environment.

Our third contribution is PerfScope, a tool for online performance bug inference. PerfScope works in two phases. First, we build a robust function-level profile to map different patterns of system calls to the functions that generate them. We then use this profile online along with unsupervised clustering to produce a ranked list of which functions are experiencing a problem at runtime. PerfScope is able to effectively identify functions experiencing a problem at runtime. However, many functions identified are false positives, that is, non-buggy functions unrelated to the bug. The last component in our framework, HSR, is triggered after PerfScope to reduce the number of false positives presented to developers.

2.4 HSR

Rule-based static analysis is a known approach to identifying buggy functions based on existing fixes to other buggy functions. However, there is a trade-off when using this approach. On one hand, rules can be written to look for very specific bugs in code. However, these rules are generally non-transferable to other systems or other bugs. On the other hand, generic rules can be written to identify functions with bug-related characteristics but which may or may not be buggy functions. However, these rules typically generate many false positives.

Our fourth contribution is HSR, a tool that combines static analysis with runtime analysis for performance anomaly diagnosis. The intuition behind HSR is that functions with buggy characteristics which are also experiencing a runtime problem are most likely related to a bug. HSR works in two phases. First, offline we use static analysis [5, 10] to create a profile of functions with buggy characteristics using several generic rules we have developed through manual analysis of existing patches. We then use this profile to dynamically filter the results of PerfScope, only outputting functions which also have buggy characteristics.
Chapter 3

UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems

In this chapter we discuss the first part of our framework, UBL. Our tool is designed to predict performance anomalies in cloud environments by monitoring black-box metrics. We begin this chapter by introducing our work followed the design and implementation details of our tool.

3.1 Introduction

Providing automatic anomaly prediction techniques that can forecast whether a system will enter an anomalous state and trigger proper preventive actions to steer the system away from the anomalous state is highly desirable. However, achieving efficient anomaly management for large-scale IaaS cloud infrastructures is a challenging task. First, applications running inside the cloud often appear as black-box to the cloud service provider. Therefore, it is impractical to apply previous white-box or grey-box anomaly detection techniques (e.g., [26]) which require application instrumentation. Second, a large-scale cloud infrastructure often runs thousands of applications concurrently. The anomaly management scheme itself must be light-weight and should operate in an online fashion. Third, it is difficult, if not totally impossible, to obtain labelled training data (i.e., measurement samples associated with normal or abnormal labels) from production cloud systems. As a result, it is hard to apply previous supervised learning techniques [38, 46, 109] for monitoring production cloud systems. More importantly, supervised
learning techniques can only detect previously known anomalies.

In this chapter, we present the design and implementation of an Unsupervised Behavior Learning (UBL) system for cloud computing infrastructures. UBL does not require any labelled training data, allowing it capture emergent system behaviors. This makes it possible for UBL to predict both known anomalies and unknown anomalies. UBL employs a set of continuous VM behavior learning modules to capture the patterns of normal operations of all application VMs. To avoid manual data labeling and capture emergent system behaviors, UBL leverages an unsupervised learning method called the Self Organizing Map (SOM) [65]. We chose the SOM because it is capable of capturing complex system behaviors while being computationally less expensive than comparable approaches such as k-nearest neighbor [108]. To predict anomalies, UBL looks for early deviations from normal system behaviors. UBL only relies on system-level metrics that can be easily acquired via the hypervisor or guest OS to achieve black-box anomaly prediction.

Specially, in this chapter we make the following contributions:

- We show how to use the SOM learning technique to achieve efficient unsupervised system behavior learning.

- We describe how to leverage the system behavior model along with the node neighborhood area size analysis to predict emergent system anomalies and infer anomaly causes.

We have implemented a prototype of UBL on top of the Xen platform [27]. We have deployed and tested UBL on the NCSU’s virtual computing lab (VCL) [15] that operates in a similar way as Amazon EC2 [1]. We conducted extensive experiments using a range of real distributed systems: 1) RUBiS, an online auction benchmark [12], 2) IBM System S, a commercial stream processing system [50], and 3) Hadoop, an open source implementation of MapReduce framework [2]. Our experimental results show that UBL can predict a range of performance anomalies with 5.9-87.7% higher true positive rates and 3.3-84.5% lower false alarm rates than other alternative schemes. UBL can achieve sufficient lead time in most cases for the system to take just-in-time preventative actions [110]. Our prototype implementation shows that UBL is feasible and imposes negligible overhead for the cloud system.

The remainder of the chapter is organized as follows. Section 3.2 presents the design details of UBL. Section 3.3 presents the experimental evaluation. Finally, Section 3.4 concludes this chapter.

### 3.2 System Design

In this section, we present the design details of the UBL system. We first describe our continuous runtime system behavior learning scheme. We then present our unsupervised anomaly
prediction algorithm that can raise advance alerts about both known and unknown anomalies. Next, we present our decentralized learning framework to achieve scalable and low-cost cloud infrastructure behavior learning.

### 3.2.1 Online System Behavior Learning

It is a challenging task to achieve efficient online system behavior learning for large-scale cloud computing infrastructures. The learning scheme first needs to achieve scalability, which can induce behavior models for a large number of application components on-the-fly without imposing excessive learning overhead. Furthermore, system metric measurements for real world distributed applications are often fluctuating due to dynamic workloads or measurement noises, which requires a robust learning scheme. We chose to use the SOM learning technique in this work to achieve scalable and efficient system behavior learning.

![SOM training process](image)

The SOM maps a high dimensional input space into a low dimensional map space (usually two dimensions) while preserving the topological properties of the original input space (i.e., two similar samples will be projected to close positions in the map). Thus, the SOM can handle multi-variant system behavior learning well without missing any representative behaviors. Specially, we collect a vector of measurements \( D(t) = [x_1, x_2, ..., x_n] \) continuously for each VM, where \( x_i \) denotes one system-level metrics (e.g., CPU, memory, disk I/O, or network traffic), and use the measurement vectors as inputs to train SOMs. UBL can dynamically induce a SOM for each VM to capture the VM’s behaviors.

A SOM is composed of a set of neurons arranged in a lattice, illustrated by Figure 3.1. Each
neuron is associated with a weight vector and a coordinate in the map. Weight vectors should be the same length as the measurement vectors (i.e., $D(t)$), which are dynamically updated based on the values of the measurement vectors in the training data. UBL uses SOMs to model system behaviors in two different phases: learning and mapping. We first describe the learning phase. We will present the mapping phase in detail in the next subsection.

During learning, the SOM uses a competitive learning process to adjust the weight vectors of different neurons. The competitive learning process works by comparing the Euclidean distance of the input measurement vector to each neuron’s weight vector in the map. The neuron with the smallest Euclidian distance is selected as the currently trained neuron. For example, Figure 3.1 shows a map consisting of 9 neurons being trained with an input measurement vector of $[0,2,4]$. We first calculate the Euclidean distance to every neuron. Neuron 1 is selected as the currently trained neuron because it has the smallest Euclidean distance to the measurement vector. That neuron’s values along with its neighbor neurons are then updated. In this example, we define our neighborhood to be the neurons in a radius of $r = 1$. Striped neurons (neurons 2, 4, and 5) are the neurons in neuron 1’s neighborhood. The general formula for updating the weight vector of a given neuron at time $t$ is given in Equation 3.1. We use $W(t)$ and $D(t)$ to define the weight vector and the input vector at time instance $t$, respectively. $N(v, t)$ denotes the neighborhood function (e.g., a Gaussian function) which depends on the lattice distance to a neighbor neuron $v$. $L(t)$ denotes a learning coefficient that can be applied to modify how much each weight vector is changed as learning proceeds.

$$W(t + 1) = W(t) + N(v, t)L(t)(D(t) - W(t))$$

(3.1)

Figure 3.1 illustrates the learning process using Equation 3.1 with a learning coefficient of 1 and a neighborhood function of $\frac{1}{4}$. We use a simple function here to illustrate the learning process, but more complex neighborhood functions are used in non-trivial applications, which we discuss further in Section 3.3. For example, neuron 2 has a weight vector of $[4,2,4]$ and the input vector is $[0,2,4]$. Taking the difference between the input vector and the weight vector gives a value of $[-4,0,0]$ which is then multiplied by 1 and $\frac{1}{4}$. This gives value of $[-1,0,0]$ which is then added to the initial weight of $[4,2,4]$ to give a final updated value of $[3,2,4]$ to neuron 2. All updated values are shown in bold. The intuition behind this approach is to make the currently trained neuron and the neurons in its neighborhood converge to the input space.

When each input vector has been used to update the map multiple times (e.g., 10 in our experiments), learning is complete. At this point, the weight vectors of neurons represent a generalization of the whole measurement vector space. Thus, the SOM can capture the normal system behaviors under different workloads. We define this phase to be the bootstrap learning phase. UBL also supports incremental updates which can continuously adjust the SOM with
new measurement vectors. However, too many incremental updates may degrade the quality of the SOM as all weight vectors may converge to a small number of vector values. This can happen when the system starts to process a completely different new workload. In this case, we can re-bootstrap the SOM with new measurement data to maintain the quality of the SOM.

When applying the SOM to learning real system behaviors, we found that UBL needs to address several metric preprocessing problems in order to achieve efficiency. First, different system metric values can have very different ranges in their raw form. For example, the MEM USAGE metric ranges from 0 to 2048, while the CPU USAGE metric expressed as a utilization percentage from 0 to 100. This is problematic for our map as large data ranges would require a large number of neurons. To address this problem, we normalize all metric values to the range \([0, 100]\) by looking at the maximum value of each metric in the learning data. We chose to normalize our values this way because we found using the absolute maximum possible value sometimes produced distorted normalized values that distribute within a small range. For example, during normal operation, the observed network traffic should be much less than the maximum traffic possible. Normalizing to the maximum possible value would mean the network traffic value would only cover a small range.

During online operation, some measurement values might exceed the maximum value in the training data. This will cause some normalized metric values to be greater than 100. However, we found this does not cause an unexpected result. By doing this, we can significantly reduce the number of neurons needed for covering the whole measurement space while still capturing the patterns of the system behavior. We also filter constant metric values which have no effect on our system to further decrease the memory footprint for storing the training data. Second, some real system metric values (e.g., memory usage in Hadoop) are highly fluctuating. We might induce a map with poor quality using the raw monitoring data. To address the problem, we apply k-point moving average filter to smooth the raw monitoring data. The length of \(k\) represents the degree of smoothing, which computes an average value for the current value with the \(k\) metric values before the current value.

Determining how to properly configure and initialize the map is critical for the performance of SOM. We first need to decide the size of the map we should use for modeling a VM’s behavior. We found a matrix topology based map with dimensions 32x32 consisting of 1024 total neurons works well for all the applications we tested. As values have been normalized to \([0, 100]\), we initialize each weight vector element to a random value between 0 and 100. We found random initialization to be necessary because initializing the weight vectors to a set of known values causes the produced map to be heavily biased towards the known values. This decreases the ability of the map to predict unknown values.

Due to the randomness used in weight vector initialization, we found the random vectors generated in some maps would only represent a subset of the training data values. This caused
only a small portion of neurons to be trained, which in turn led to a poor quality map. To address this problem, we use K-fold cross validation as part of our learning phase, which works as follows. The training data is first partitioned into K parts denoted by $D_1, \ldots, D_K$. The validation process takes K rounds to complete. In round $i$, $1 \leq i \leq K$, $D_i$ is selected to be the testing data while the other $(K - 1)$ parts $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_K$ are used as the training data. We collect various correct and incorrect classification statistics to compute the accuracy of each map. Since UBL is designed to be unsupervised, we only use unlabeled normal data to train the map. UBL relies on the SLO feedback from the application or some external SLO monitoring tool [30] to select normal data. Suppose $N_{fp}$ is the number of false positives, when UBL raised an alarm yet no anomaly was found. $N_{fn}$ is the number of false negatives, when UBL failed to raise an alarm but the current sample was an anomaly. $N_{tp}$ is the number of true positives, when UBL raised an alarm and there was an anomaly. $N_{tn}$ is the number of true negatives, when UBL did not raise an alarm and the current sample was normal. Since our training data are all normal data, $N_{fn} = N_{tp} = 0$. The accuracy metric for each map is calculated using the standard way as follows:

$$A = \frac{N_{tn} + N_{tp}}{N_{tn} + N_{fp} + N_{fn} + N_{tp}}$$

(3.2)

The cross validation module selects the map with the best accuracy as the final trained map. We use the same Gaussian neighborhood function and the same constant learning coefficient among all datasets. We also conducted sensitivity experiments to show how those parameter values affect the performance of UBL. We will present those results in Section 3.3.

### 3.2.2 Unsupervised Anomaly Prediction

Performance anomalies, such as SLO violations, in distributed systems often manifest as anomalous changes in system-level metrics. Faults do not always cause a SLO failure immediately. Instead, there is a time window from when the fault occurs to the actual time of failure. Therefore, at any given time, a system can be thought to be operating in one of three states: normal, pre-failure, or failure. Additionally, the system typically first enters the pre-failure state before entering the failure state. Since the SOM is able to maintain the topological properties of the measurement samples, we can observe when the system enters the pre-failure state and moves to the failure state. Figure 3.2 shows an example using a real system failure where the failing system follows a path through the SOM over time. UBL can raise an advanced alarm when the system leaves the normal state but has not yet entered the failure state.

To decide the system state represented by each neuron, UBL calculates a neighborhood area size for each neuron in the SOM.
As mentioned in Section 3.2.1, when neurons in the SOM are updated with training data, we also adjust the weight vectors of their neighboring neurons. After learning, frequently trained neurons will have modified the weight vector values of their neighboring neurons with the same input measurement vectors. As a result, the weight vectors of the neurons that are frequently trained will look similar to the weight vectors of their neighboring neurons. Since systems are usually in the normal state, neurons representing the normal state will be more frequently trained than the neurons representing the pre-failure or failure states. Thus, we will have clusters of neurons representing different normal system behaviors. We calculate a neighborhood area size value for each neuron by examining the immediate neighbors of each neuron. As our lattice topology is a two-dimensional grid, this means we examine the top, left, right, and bottom neighbors. We calculate the Manhattan distance between two neurons $N_i, N_j$ with weight vectors $W_i = [w_{1,i}, \ldots, w_{k,i}], W_j = [w_{1,j}, \ldots, w_{k,j}]$ respectively, as follows:

$$M(N_i, N_j) = \sum_{l=1}^{k} |w_{l,i} - w_{l,j}|$$

(3.3)

We define the neighborhood area size for a neuron $N_i$ as the sum of Manhattan distance between the neuron $N_i$ and its top, left, right and bottom immediate neighbors denoted by $N_T, N_L, N_R$ and $N_B$, as follows:

$$S(N_i) = \sum_{X \in \{N_T, N_L, N_R, N_B\}} M(N_i, X)$$

(3.4)
Figure 3.3: Grey-scale visualization of the SOM models for the RUBiS with the Network Hog fault and System S with the MemLeak fault. Darker neurons have larger neighborhood area sizes while lighter neurons have smaller neighborhood area sizes.

UBL determines if a neuron is normal or anomalous by looking at the neighborhood area size of that neuron. If the neighborhood area size is small, we know that the neuron we have mapped to is in a tight cluster of neurons, meaning the neuron is normal. On the other hand, if a neuron maps to a neuron with a large neighborhood area value, we know that the neuron is not close to other neurons, and thus, probably anomalous. For example, in Figure 3.2, the calculated neighborhood area size for neuron 6 (a normal neuron) of would be the sum of the differences to neighbors 2, 5, 7, and 10, which is 102. The neighborhood area size of neuron 10 (a pre-failure neuron), on the other hand, is the sum to neighbors 6, 9, 14, and 11, which is 280.

Figure 3.3 shows two maps after bootstrap learning has completed: one is for the RUBiS web server with a network hog bug and the other is for one faulty component in System S including a memory leak bug. We use gray-scale visualization to illustrate the behavior patterns. Darker neurons represent anomalous behaviors while lighter neurons represent normal behaviors. Once learning is complete, we can clearly see different systems present distinct behavior patterns that can be captured by the SOM.

During application runtime, we map each measurement vector to a neuron using the same Euclidean distance metric as the learning phase. We look at the neighborhood area size of the mapped neuron. If the neighborhood area size is below the threshold for the map, that means the sample has mapped to a neuron which is close to many other neurons. We consider this sample to be a normal sample and do not raise an alarm. However, if the sample maps to a neuron with an area value greater than or equal to our threshold value, this sample represents something we rarely see during learning. We consider this type of sample to be anomalous. Transient fluctuations in system metrics due to noise can still be present even after data smoothing. Those momentary fluctuations may be mapped to anomalous neurons, although it would be
incorrect to raise an alarm in this case. As a result, we raise an alarm only when the system identifies three consecutive anomalous samples.

Determining a neighborhood area size threshold to differentiate normal and anomalous neurons is integral to the accuracy of the UBL system. If the threshold is set too high, we cannot raise an alarm early enough and may miss some anomalies. Alternatively, if we set the threshold too low, we might raise too many alarms, including false alarms. Additionally, neighborhood area size values vary from map to map depending on the range of values in the dataset. To address this issue, we set the threshold value based on a percentile instead of a fixed value. We sort all calculated neighborhood area size values and set the threshold value to be the value at a selected percentile. We found a percentile value of 85% is able to achieve good results across all datasets in our experiments. We further examine the effect of the threshold on accuracy in Section 3.3.

3.2.3 Anomaly Cause Inference

Determining the root cause of an anomaly is a highly non-trivial task. UBL is able to ameliorate this task by giving a hint as to what metrics are the top contributors to an anomaly. While this does not directly identify the root cause of the anomaly, it provides a clue of where to start looking. As the SOM preserves the topological properties of the measurement space, UBL can use this information to identify the faulty metric causing an anomaly. The basic idea is to look at the difference between anomalous neurons and normal neurons, and output the metrics that differ most as faulty metrics. Specifically, when we map a measurement sample to an anomalous neuron, we calculate the Euclidean distance from the mapped anomalous neuron to a set of nearby normal neurons. Here, it is necessary to avoid comparing with anomalous neighbor neurons as they represent unknown states and therefore may give incorrect anomaly cause hints. We examine the neighborhood area value for each neuron first. If it is above our threshold, we ignore it and move on to the next neuron in our neighborhood. If no normal neuron is found in the anomalous neuron’s neighborhood, we expand our distance calculation to include more neurons in the map. In order to ensure we get a good representation of normal metrics, we select Q normal nearby neurons (e.g., Q = 5 in our experiments).

Once a set of normal neurons has been found, we calculate difference between the individual metric values of each normal neuron and those of the anomalous neuron. As the change can be positive or negative, we take the absolute value of the calculated difference. We then sort the metric differences from the highest to the lowest to determine a ranking order. After this process completes, we will have Q metric ranking lists. Finally, we examine the ranking orders of each of the Q rankings to determine a final order. To do this, we use majority voting. Each list votes for which metric it had identified as having the largest difference in values. We then
output the metric with the most votes as the first ranked metric, the metric that has the 2nd most is the second ranked metric, and so on. Ties indicate no consensus could be reached and we output the metric that happens to come first in the output list construction. While we have found ties to be rare, a potential refinement of this approach would be to use the total difference of each metric to break ties. As an example, suppose three ranking lists rank CPU usage as the top anomaly cause but two other ranking lists rank Memory usage as the top cause. We will output CPU usage as the top anomaly cause as it has been ranked the top anomaly cause by a majority.

### 3.3 Experimental Evaluation

We have implemented a prototype of UBL on top of the Xen platform and conducted extensive experiments using three benchmark systems: the RUBiS multi-tier online auction web application (EJB version) [12], IBM System S data stream processing system [50], and the Hadoop MapReduce framework [2]. We begin by describing our evaluation methodology. We then present our results.

#### 3.3.1 Evaluation Methodology

Our experiments were conducted on the Virtual Computing Lab (VCL) infrastructure [15] which operates in a similar way as Amazon EC2 [1]. Each VCL host has a dual-core Xeon 3.0GHz CPU and 4GB memory, and runs 64bit CentOS 5.2 with Xen 3.0.3. The guest VMs also run 64bit CentOS 5.2.

UBL monitors VMs’ resource demands from domain 0, using the libxenstat and libvirt libraries to collect resource usage information (e.g., CPU usage, memory allocation, network I/O, disk I/O) for both domain 0 and guest VMs. UBL also uses a small memory monitoring daemon within each VM to get memory usage statistics (through the /proc interface in Linux). The sampling interval is 1 second.

We have chosen three benchmark systems to evaluate UBL in order to demonstrate the agnosticism necessary for such a system to be used in the real world. Moreover, UBL can handle dynamic applications processing time-varying workloads. To demonstrate this, we drive all the benchmark applications using dynamic workload intensity observed in real world online services. We injected faults at different times while the system was under dynamic workload. Each experiment duration varies slightly but all last about one hour. Fault injections also vary slightly depending on the fault type but all last between 1 and 5 minutes. For each fault injection, we repeated the experiment 30 to 40 times. We now describe all the systems and fault injections in detail as follows.
**RUBiS online auction benchmark:** We used the three-tier online auction benchmark system RUBiS (EJB version) with one web server, two application servers, and one database server. In order to evaluate our system under workloads with realistic time variations, we used a client workload generator that emulates the workload intensity observed in the NASA web server trace beginning at 00:00:00 July 1, 1995 from the IRCache Internet traffic archive [14] to modulate the request rate of our RUBiS benchmark. The client workload generator also tracks the response time of the HTTP requests it made. A SLO violation is marked if the average request response time is larger than a pre-defined threshold (e.g., 100ms).

We injected the following faults in RUBiS: 1) *MemLeak*: we start a memory-intensive program in the VM running the database server; 2) *CpuLeak*: a CPU-bound program with gradually increasing CPU consumptions competes CPU with the database server inside the same VM; and 3) *NetHog*: we use httperf [8] tool to send a large number of http requests to the web server.

**IBM System S:** We used the IBM System S that is a commercial high-performance data stream processing system. Each System S application consists of a set of inter-connected processing elements (PEs). We measured the average per-tuple processing time. A SLO violation is marked if the average processing time is larger than a pre-defined threshold (e.g., 20ms). In order to evaluate our system under dynamic workloads with realistic time variations, we used the workload intensity observed in the ClarkNet web server trace beginning at 1995-08-28:00.00 from the IRCache Internet traffic archive [14] to modulate the data arrival rate.

For System S, we injected the following faults: 1) *MemLeak*: we start a memory-intensive program in one randomly selected PE; 2) *CpuHog*: a CPU-bound program competes CPU with one randomly selected PE within the same VM; and 3) *Bottleneck*: we make one PE the bottleneck in the application by setting a low CPU cap for the VM running the PE.

**Hadoop:** We run Hadoop sorting application that is one of the sample applications provided by the Hadoop distribution. We measure the progress score of the job through Hadoop API. A SLO violation is marked when the job does not make any progress (i.e., 0 progress score increase). We use 3 VMs for Map tasks and 6 VMs for Reduce tasks. The number of map slots on each VM running map tasks is set to 2, and the number of Reduce slots on each VM running reduce tasks is set to 1. We use this configuration because the reduce task requires much more disk and memory space than the map task in the sorting application. Since this is a small Hadoop cluster, the JobTracker and NameNode are very light-weight. We colocate them together with the first reduce VM. The data size we process is 12GB, which is generated using the RandomWriter application.

For Hadoop, we injected two types of faults into all the VMs running the map tasks: 1) *MemLeak*: we injected a memory leak bug into all the map tasks, which repeatedly allocates certain memory from the heap without releasing; and 2) *CpuHog*: we injected an infinite loop bug into all the map tasks.
We evaluate the anomaly prediction accuracy using the standard receiver operating characteristic (ROC) curves. ROC curves can effectively show the tradeoff between the true positive rate ($A_T$) and the false positive rate ($A_F$) for a prediction model. We use standard true positive rate $A_T$ and false positive rate $A_F$ metrics given in equation 3.5. The $N_{tp}$, $N_{fp}$, $N_{tn}$, and $N_{fn}$ values are the same as those described in Section 3.2.

$$A_T = \frac{N_{tp}}{N_{tp} + N_{fn}}, \quad A_F = \frac{N_{fp}}{N_{fp} + N_{tn}}$$

We say the prediction model makes a true positive prediction if it raises an anomaly alert at time $t_1$ and the anomaly indeed happens at time $t_2$, $t_1 < t_2 < t_1 + W$, where $W$ denotes the upper-bound of the anomaly pending time.$^1$ Otherwise, we say the prediction model fails to

---

$^1$We have determined an appropriate anomaly pending time upper-bound $W$ for each dataset by manually examining the fault injection time to the SLO violation time. For example, if a fault is injected at time $t = 20$ and a SLO violation is observed at time $t = 30$, our window size would be 10.
make a correct prediction. If the predictor raises an alert and the predicted anomaly does not happen within the $t_1 + W$, we say that the prediction model raises a false alarm. We further evaluate the prediction capability of UBL using *achieved lead time*, which we define to be the amount of lead time we give prior to a SLO violation occurring. For example, if we raise an alarm at time $t$ and the actual SLO violation occurs at time $t + 20$ seconds, we have achieved a lead time of 20 seconds.

For comparison, we also implemented a set of commonly used unsupervised learning schemes: 1) the *PCA* scheme uses principle component analysis to identify normal and anomalous samples [87]; and 2) the *k-NN* scheme calculates a k-nearest neighbor distance for each measurement sample to identify normal and anomalous samples [108]. Different from UBL, both PCA and k-NN models need to be trained with both normal and anomalous data. In contrast, UBL does not require the training data to contain anomalous data. We use *UBL-NS* to denote the UBL scheme without applying any data smoothing. We use *UBL-$kPtS$* to represent the UBL scheme using the $k$-point moving average smoothing. Through experimentation, we have defined our map to be 32x32 nodes, the neighborhood of each node to have a radius of 4, the learning factor to be a constant 0.7, and the neighborhood function to be a Gaussian function. We use 3-fold cross validation to select the best map among three randomly initialized map. We have also conducted sensitivity study experiments on those parameters, which will be presented in the next subsection.

### 3.3.2 Results and Analysis

**Prediction Accuracy Results**

We now present the anomaly prediction accuracy comparison results. We acquire the ROC curves for the UBL schemes by adjusting the neighborhood area size percentile threshold (i.e., 70'th percentile to 98'th percentile). For PCA, we obtain the ROC curves by adjusting the
variance threshold. The ROC curves of k-NN is calculated by adjusting the $k^{th}$ nearest neighbor distance threshold.

We begin with the results of our RUBiS experiments. Figure 3.4 shows the ROC curves for the RUBiS systems under three different faults. The memory leak dataset was our best RUBiS dataset, we were able to achieve a high true positive rate of 97% with a very low false positive rate of 2%. This is consistent with what we expected as the memory leak manifests gradually and slowly. Conversely, we see our worst results from our NetHog dataset, achieving a 87% maximum true positive rate with a corresponding 4.7% false positive rate. This is also consistent with what we expected since the NetHog fault manifests more quickly than the other two faults. In all cases, UBL consistently outperforms PCA and k-NN with higher true positive rates and lower false positive rates.

We can see the positive effect of smoothing by looking at the Memleak dataset. Due to the gradual nature of this fault, smoothing allows us to achieve approximately 20% higher true positive rates with corresponding false positive rates. This is expected as the RUBiS dataset contains quite some transient noises. Additionally, due to the gradual manifestation time of the fault, we do not smooth out any pre-failure symptoms. Therefore, we see a marked improvement between the smoothed and non-smoothed data.

Figure 3.5 shows the prediction accuracy results for the IBM System S application under different faults. The results show that UBL is able to achieve higher prediction accuracy than the other schemes in all cases. The best result we were able to achieve was a 98% true positive rate along with a 1.7% false positive rate in the memory leak dataset. The worst results we achieved were in the CPU Hog dataset, with a 93% true positive rate and a 0.5% false positive rate. This is expected as CPU spikes are more difficult to predict due to the rapid onset of the fault. Similarly, the Bottleneck fault is also hard to predict as the time from fault to failure is also short. The System S dataset has relatively less noise than the RUBiS datasets, so the high accuracy results are expected. Additionally, the Bottleneck and CPU Hog datasets are harder
to predict than the Memleak dataset, while our results for these datasets are good, they are lower than Memleak as expected.

It is interesting to observe smoothing does not always help achieve better accuracy. In the Bottleneck and CPU Hog datasets, the best results we achieve are those without any smoothing. This is due to two reasons. First, both faults manifest very quickly. Second, System S datasets are inherently not very noisy. When we apply smoothing, even 5-point smoothing, we sometimes smooth out those critical pre-anomaly symptoms. When this happens, our model is unable to raise an alarm appropriately, leading to lower accuracy than without smoothing.

We now present the results of our Hadoop experiments shown by Figure 6. The MemLeak dataset was able to achieve the highest overall true positive rate due to the gradual nature of the fault. Hadoop is our noisiest dataset, which explains for the high false positive rates observed in these datasets. As expected, the rapid onset time of the CpuHog fault means the overall true positive rate we could achieve here was lower than the gradual memory leak dataset. As we can see, smoothing helps the MemLeak dataset, reducing the overall noise of the dataset while preserving the pre-failure symptoms. We show an additional curve to illustrate this point. Conversely, while smoothing reduced the noise of the CpuHog dataset, reducing the overall false positive rate, it also smoothed out pre-failure symptoms leading to a lower true positive rate as well. In both cases, UBL still can achieve better prediction accuracy than PCA and k-NN.

**Lead Time Results**

Figure 7 shows the average lead times achieved by UBL for RUBiS, System S, and Hadoop, respectively. The results shown only consider the lead time achieved for cases determined to be true positive results. We first discuss the RUBiS lead time results. We were closest to the maximum achievable lead time in the CpuLeak dataset. We achieved an average lead time of 38 seconds, with a maximum lead time of 40 seconds. The memory leak results for this dataset were the worst results we saw. We achieved an average lead time of only 7 seconds, with a maximum lead time of 50 seconds. This can be explained by variations in the data. The workload and background noise of the system caused the metrics to approach unknown levels only when the system was close to the anomaly state.

We next discuss the lead time we were able to achieve for the System S datasets. Here, we were able to achieve an average lead time of 47 seconds for the memory leak dataset with a maximum lead time possible of 50 seconds. While the lead time is lower for the CpuHog dataset, we achieved an average lead time of 3 seconds, with a maximum possible lead time of 4 seconds. Similarly, we achieved a lead time of 5 seconds in the Bottleneck dataset, with a maximum lead time of 6 seconds possible. The memory leak dataset had the best lead time because it was a gradual change with little memory fluctuation. The CpuHog and BottleNeck datasets had much
shorter manifestation durations, and thus our system had little time to predict the anomaly, however we still are able to achieve results close to the maximum possible lead time.

Finally, we present the average lead time we are able to achieve in the Hadoop experiments in. The average lead time we were able to achieve in the memory leak dataset was 24 seconds. Here the maximum lead time possible was 25 seconds. In this case, UBL is able to quickly determine the pattern is not normal and raise an early alarm. In contrast, the CpuHog lead time is lower, but as before this is due to the rapid onset time. We achieved an average lead time of 3 seconds with a maximum possible lead time of 4 seconds.

All in all, UBL can achieve close to maximum possible lead time for different faults tested in our experiments. Our previous study [110] shows that we can take local anomaly prevention actions such as VM resource scaling within one second and more costly anomaly preventions such as live VM migration within 10 to 30 seconds. Thus, the lead time achieved by UBL is sufficient in most cases for the cloud system to provide automatic anomaly preventions.

Anomaly Cause Inference Results

We now present our anomaly cause inference results shown by Figure 3.8. We consider the faulty metric to be the metrics most closely associated with a given failure. For example, for the MemLeak datasets, we consider the memory metric as the faulty metric. The figure shows the ranking of the faulty metric in the rank list output by UBL.

![Figure 3.8: Ranking results of the faulty metrics in different failure instances. The Y axis is the faulty metric rank as determined by UBL while the X axis represents the total number of faults observed.](image)

As the Figure shows, UBL can correctly rank the faulty metric as top ranked metric in most failure cases. These results indicate UBL is able to preserve the topological properties of the input measurement space and is useful for diagnosis as well as prediction. In datasets where
Table 3.1: UBL System overhead measurements.

<table>
<thead>
<tr>
<th>System Modules</th>
<th>CPU cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM monitoring (8 attributes)</td>
<td>1.33 ± 0.09 ms</td>
</tr>
<tr>
<td>3-fold cross validation (6000 samples)</td>
<td>42 ± 1 sec</td>
</tr>
<tr>
<td>SOM model updating</td>
<td>245 ± 54.9 ms</td>
</tr>
<tr>
<td>Anomaly prediction</td>
<td>2.4 ± 2.6 ms</td>
</tr>
</tbody>
</table>

Table 3.2: Sensitivity experiment results for the NetHog fault in RUBiS and MemLeak fault in System S.

<table>
<thead>
<tr>
<th></th>
<th>Accuracy (NetHog)</th>
<th>Accuracy (MemLeak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map 25x25</td>
<td>97%</td>
<td>93%</td>
</tr>
<tr>
<td>Map 32x32</td>
<td>98%</td>
<td>92.8%</td>
</tr>
<tr>
<td>Map 40x40</td>
<td>97.1%</td>
<td>93.3%</td>
</tr>
<tr>
<td>Neighborhood size 3</td>
<td>98.6%</td>
<td>93.6%</td>
</tr>
<tr>
<td>Neighborhood size 4</td>
<td>98.5%</td>
<td>92.8%</td>
</tr>
<tr>
<td>Neighborhood size 5</td>
<td>97.6%</td>
<td>92.5%</td>
</tr>
<tr>
<td>Gaussian function height 7</td>
<td>98.2%</td>
<td>92.9%</td>
</tr>
<tr>
<td>Gaussian function height 10</td>
<td>98.5%</td>
<td>92.8%</td>
</tr>
<tr>
<td>Gaussian function height 13</td>
<td>98.9%</td>
<td>93.8%</td>
</tr>
</tbody>
</table>

noise is less of an issue, such as System S, UBL achieves near perfect ranking results.

**Overhead Results**

We now discuss the overhead of the UBL system. Table 3.1 lists the CPU cost of each key module in our system. The VM monitoring module runs within Domain 0 of each host and collects eight resource attributes per second. Each collection takes about 1.3 milliseconds. 3-fold cross validation is the most time-consuming operation, taking about 42 seconds. However, this step is only used during bootstrap learning phase. Incremental SOM updates take about 245 milliseconds for every 30 new data samples. Anomaly prediction takes about 2.4 milliseconds. During the normal execution, the learning VM imposes less than 1% CPU load and UBL consumes less than 16MB of memory. Overall, the overhead measurements show that UBL is light-weight, which makes it practical for online system anomaly management.

**Sensitivity Study**

We have conducted sensitivity experiments to study how UBL performs under different key parameter settings. We show our results in Table 3.2. The accuracy values are calculated using Equation 3.2. We observe that UBL is not very sensitive to different parameter values and is able to achieve accuracy values which differ by less than 1% in most cases. The map size parameter has the potential to affect the accuracy of the system if it is set too low. For example,
a 5x5 map is too small to effectively capture the overall pattern of the system. Additionally, if the map size is too large, the learning time becomes long. We have found map sizes in the range we list are able to give good results for all datasets we tested.

### 3.4 Summary

In this chapter, we have presented UBL, a novel black-box unsupervised behavior learning and anomaly prediction system for IaaS clouds. UBL leverages the Self-Organizing Map (SOM) learning technique to capture dynamic system behaviors without any human intervention. Based on the induced behavior model, UBL can predict previously unknown performance anomalies and provides hints for anomaly causes. We have implemented a prototype of UBL on top of the Xen platform and conducted extensive experiments using real world distributed systems running inside a production cloud infrastructure. Our results show that UBL can achieve high prediction accuracy with up to 98% true positive rate and 1.7% false positive rate, and raise advance alarms with up to 47 seconds lead time. UBL is light-weight, which makes it practical for large-scale cloud computing infrastructures.
Chapter 4

Online Faulty Function Localization for Debugging Performance Anomalies in Cloud Hosting Infrastructures

In this chapter we discuss the design and evaluation of the second component in our framework, PerfCompass. When an alert is raised by an anomaly prediction system, PerfCompass is triggered perform fine-grained fault localization in cloud environments. We begin by introducing the work, then discuss PerfCompass in detail.

4.1 Introduction

When a performance anomaly occurs, it is important to identify the root cause and correct the performance anomaly quickly in order to avoid significant financial penalties for both the cloud service provider and the cloud user. However, it is challenging to diagnose performance anomalies in a production cloud infrastructure is a challenging task. Due to the multi-tenancy and resource sharing nature of IaaS clouds, it is often difficult to identify the root cause of a production-run performance anomaly. On one hand, a performance anomaly fault can have a global impact, affecting almost all the executing threads of an application when the fault is triggered. For example, a VM with insufficient resources (e.g., insufficient memory) will cause all executing threads of an application to be affected. On the other hand, a fault can also have a local impact, affecting only a subset of the executing threads of an application right after the fault is triggered. For example, an infinite loop bug reading from a socket will only directly affect
the threads executing the infinite loop directly (e.g., increasing the frequency of `sys_read`). External faults such as interference from other co-located applications or improper resource allocation typically have a global impact on the system. As identified by previous work, both external environment issues [135] and internal software bugs [70] are major problems existing in the cloud. It is critical to identify whether a fault has a global or local impact on the system as the steps taken to diagnose and correct those two different kinds of faults are quite different. When we find a fault has a global impact we can try a simple fix first such as alerting the infrastructure management system to migrate the application to a different host. If the fault has a local impact, however, the cause is likely to be an internal software bug, requiring the application developer to diagnose and fix it.

Although previous work [17, 25, 33, 34, 48, 54, 83, 95–97, 128] has studied the online fault localization problem under different computing contexts, existing approaches can only provide coarse-grained fault localization, that is, identifying faulty application components. However, they cannot distinguish between faults with a global or local impact since both global and local faults may manifest similarly at the application component level (e.g., increased CPU usage). Additionally, tools such as Fay [47] and DARC [111] can help developers debug the root cause of performance anomalies. However, these tools are difficult to adapt to a production cloud environments due to the large overhead they impart or application instrumentation they require.

In this chapter, we present PerfCompass, an online fine-grained performance anomaly fault localization and inference tool designed to differentiate faults with a global impact from faults with a local impact and provide diagnostic hints for either cloud system administrators or cloud users. Figure 4.1 illustrates the overall structure of the PerfCompass system. PerfCompass employs lightweight kernel-level system call tracing to continuously record system calls from

![Figure 4.1: PerfCompass overview.](image-url)
the monitored application. PerfCompass performs fault localization and inference using four steps. First, we segment the large raw system call traces into groups of closely related system calls which we call execution units. Next, we process these execution units to extract a set of fine-grained fault features (e.g., which threads are affected by the fault, how quickly the fault manifests in different threads). Third, we use these fine-grained fault features to differentiate between faults with a global impact and faults with a local impact. PerfCompass then provides diagnostic hints by suggesting the root cause of the fault as being an external environment issue or internal software bug. Lastly, PerfCompass identifies the top affected system calls and provides ranks for those affected system calls. Knowing what system calls are affected by the fault allows developers to identify which subsystems (e.g., network, I/O, CPU) are affected and gives hints on how to perform further diagnosis on the fault.

In this chapter, we make the following contributions:

- We develop a novel fine-grained fault feature extraction algorithm that can identify which threads are affected by the fault and how quickly the fault manifests in different threads.
- We describe how external environment issues typically have a global impact on the system while internal software bugs can have a global or local impact on the system. We show how to use this information to localize performance anomalies as external or internal.
- We present a system call ranking algorithm which identifies the top affected system calls and provides ranks for those affected system calls. This system call ranking is useful for system administrators or users to gain insight about the root cause of the performance anomaly.
- We describe a robust execution unit extraction scheme that can properly split a large raw system call trace into fine-grained groups of closely related system calls for different kinds of applications.
- We have implemented PerfCompass and conducted an experimental study using five open source systems (Apache, MySQL, Tomcat, Cassandra, and Hadoop). The results show that PerfCompass can correctly localize 23 out of the 24 tested faults (17 common environment issues and 7 real software bugs) without calibration and achieve 100% accuracy with calibration. Furthermore, PerfCompass provides useful hints for both external and internal faults. Our approach is lightweight, imparting an average of 2.1% runtime overhead to the tested server applications.

The rest of the chapter is organized as follows. Section 4.2 discusses the preliminaries of our system. Section 4.3 describes the design of the PerfCompass system. Section 4.4 presents our experimental evaluation. Finally, we conclude the chapter in Section 4.5.
4.2 Preliminaries

In this section, we first describe our system model for IaaS clouds. We then describe our problem formulation followed by the key assumptions we make.

4.2.1 System Model

IaaS clouds are comprised of several physical hosts connected by networks. These physical hosts use virtualization technology (e.g., KVM [64], Xen [27]) to run several different virtual machines (VMs), which are then leased to end users. We instrument each VM with a lightweight kernel tracing tool called the Linux Trace Toolkit Next Generation (LTTng) [43] which continuously monitors the system calls generated by each application running on that VM.

The collected system call traces are then stored on a globally accessible NFS server. PerfCompass is decoupled from the VMs it monitors and only needs access to the system call trace of a monitored VM to perform its runtime analysis. Thus, PerfCompass can be encapsulated in special analysis VMs which could be dynamically placed on lightly loaded hosts to use the residual resources in the IaaS cloud for fault localization.

PerfCompass is triggered when an alarm is raised by an online performance anomaly detection tool [40, 110]. Fault localization is performed on the faulty components identified by existing online component-level fault localization tools (e.g., [34, 83]).

Table 4.1 shows the various terms we have defined and is intended as a reference to help avoid any ambiguity while discussing how each item is used.

4.2.2 Problem Formulation and Key Assumptions

The goal of PerfCompass is to provide two key pieces of information: 1) the performance anomaly impact results; and 2) performance anomaly root cause hints. Specifically, PerfCompass first quantifies the impact of a detected fault as either global or local, suggests the cause as external or internal, and then identifies the system calls which are most affected by the fault.

PerfCompass is designed specifically for performance anomalies which typically provide little information (e.g., no error messages, limited stack trace) for debugging. However, our work focuses on the anomalies which are caused by either infrastructure sharing issues or software bugs. Performance anomalies due to an application misconfiguration have been studied before [22] and are beyond the scope of this work.

PerfCompass targets multi-threaded or multi-processed server applications which are prone to performance anomalies. We observe that most modern server applications are multi-threaded or multi-processed in order to utilize the multi-core processors efficiently.

PerfCompass aims at providing fast online fault debugging and root cause inference within
Table 4.1: Notations.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>The type of system call (e.g., sys_write)</td>
</tr>
<tr>
<td>$t_i$</td>
<td>The timestamp of system call $s_i$</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>An execution unit (closely related system calls)</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>The fault onset time (how fast a fault effects a thread)</td>
</tr>
<tr>
<td>$\tau$</td>
<td>The fault impact factor (percentage of affected threads over total threads)</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>The fault onset time dispersion (fault onset time standard deviation among all affected threads)</td>
</tr>
<tr>
<td>$k$</td>
<td>The moving average window length</td>
</tr>
<tr>
<td>$C$</td>
<td>The system call frequency count</td>
</tr>
<tr>
<td>$T$</td>
<td>The time interval of start of execution unit to system call $s_i$</td>
</tr>
<tr>
<td>$W$</td>
<td>The buffered window of recent system calls</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>The fault onset time threshold</td>
</tr>
<tr>
<td>$\beta$</td>
<td>The fault onset time dispersion threshold</td>
</tr>
<tr>
<td>$\delta$</td>
<td>The external fault impact factor threshold</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>The internal fault impact factor threshold</td>
</tr>
<tr>
<td>$\Theta$</td>
<td>The threshold for system call execution time and system call frequency outlier detection</td>
</tr>
</tbody>
</table>

the production cloud infrastructure. Thus, PerfCompass needs to be light-weight without imposing high overhead to the production system. Note that PerfCompass is not designed to support detailed software debugging such as localizing root cause buggy functions, which require application knowledge and developer involvement. However, we believe that PerfCompass can avoid unnecessary software debugging effort by quantifying the fault impact as global or local, suggesting the root cause as either external or internal, and providing runtime fault debugging hints.

4.3 System Design

In this section, we present the design details of the PerfCompass system. We begin with an overview of our system. We then describe each component of PerfCompass. Lastly, we describe two enhancements which can help improve the fault localization accuracy.
4.3.1 System Overview

PerfCompass performs fault localization and inference using four components: 1) execution unit extraction; 2) fault onset time identification; 3) fault differentiation; and 4) affected system call ranking. PerfCompass first quantifies the fault impact as either global or local. PerfCompass next suggests root cause of the fault as either external or internal. Finally, PerfCompass identifies the system calls affected by the fault and ranks them in order to provide debugging clues to developers and system administrators.

Kernel-level system call tracing allows us to monitor application behavior without imparting significant overhead. However, the system call tracing tool typically produces a single large system call trace for each application. It is infeasible to perform fine-grained fault localization using the raw system call trace directly. Hence, PerfCompass first employs an execution unit extraction component to segment the raw system call traces into fine-grained groups of closely related system calls called execution units.

Next, we analyze the different extracted execution units to answer the following two questions: 1) which threads are affected by the fault? and 2) how quickly are they affected by the fault? To do this, we first compute a fault onset time metric to quantify how quickly a thread is affected by the fault. We then extract a fault impact factor metric, which computes the percentage of threads affected by the fault. We also extract a fault onset time dispersion metric, which quantifies the differences among the fault onset time values of the affected threads.

Third, we use the fault features to determine whether the fault has a global impact on the system or a local impact on the system. Intuitively we observe that external faults have global impact on the system while internal faults can have either a global or local impact. The reason is that an external fault, such as an insufficient CPU allocation, will directly affect all running threads regardless of what code they are executing. In contrast, only those threads executing the buggy code are directly affected by an internal fault with other threads being affected indirectly due to shared resources or inter-thread communication. Depending on the application logic, this means the fault could have a global or local impact on the system. We use this observation to suggest the root cause of the fault as either external or internal.

Lastly, PerfCompass identifies any system calls showing a significant increase in either execution time or frequency, providing a ranked list of these affected system calls based on the observed increase. Those top affected system calls can provide important hints about both external and internal faults. For example, when a co-located VM causes disk contention, I/O related system calls (e.g., \texttt{sys\_read}, \texttt{sys\_write}) will be more affected the other types of system calls. Similarly, when an internal bug causes an infinite loop in a blocking I/O function, we would expect to see an increase in the frequency of blocking I/O related system calls (e.g., \texttt{sys\_futex}, \texttt{sys\_write}).
4.3.2 Execution Unit Extraction

PerfCompass uses LTTng to collect system calls (denoted by $s_i$) issued by the monitored application from the guest OS. Each system call entry consists of 1) \{timestamp 1, process/thread ID, system call name\} and 2) \{timestamp 2, process/thread ID, $\text{sys\_exit}$\}. We derive the system call execution time by computing the difference between timestamp 1 and timestamp 2.

We buffer a window (e.g., $W = 1.5$ minutes) of recent system calls in the kernel buffer. Older system call data are written into a NFS server in case longer trace-data are needed for fault localization. When a performance anomaly is detected, PerfCompass is triggered to perform online fault localization and inference using the buffered trace. Typically, the buffered data is sufficient for PerfCompass to perform fault localization as most faults manifest within a short period of time. However, if the buffered data are insufficient, PerfCompass can retrieve additional trace data from the remote NFS server.

The kernel tracing tool produces a single stream of system calls generated by all threads and processes for each application. In order to perform fine-grained fault localization, we need to segment this large raw trace into execution units. We first group system calls based on thread IDs. The rationale for doing this is that different threads are typically assigned with different tasks (e.g., logging, request processing). Therefore, the system calls generated by the individual threads are likely to be related to each other, though not necessarily from the same code block.

When processing the thread-based execution units, we found that system calls separated by relatively large time gaps were being grouped together into the same execution unit. These time gaps are problematic, as they give skewed fault onset time values, leading to incorrect fault localization results. Specifically, grouping two contiguous system calls with a large time gap into one execution unit causes us to derive an inflated fault onset time value for that execution unit. When exploring this issue, we found that a large time gap between two contiguous system calls can be caused by two reasons: 1) thread reuse, and 2) non-deterministic thread scheduling. For example, we observed that some server systems (e.g., Apache web server) use a pre-allocated thread pool in which the threads are reused for different tasks to avoid the overhead of dynamically creating new threads. We also found a context switch, invoked by the CPU scheduler, could occur between two contiguous system calls in the same application, causing a large time gap.

To address this problem, we further divide each per-thread execution unit based on the time gaps between two contiguous system calls to mitigate the inaccuracies caused by thread recycling or the CPU scheduler. Specifically, if the time gap between two contiguous system calls $s_1$ and $s_2$ is larger than a certain threshold, we say the current execution unit $\lambda_1$ ends...
at \( s_1 \) and a new execution unit \( \lambda_2 \) begins at \( s_2 \). This ensures that each execution unit does not contain two system calls separated by a gap larger than the threshold. However, setting a proper threshold is a challenging task. In this work, we use the fault onset time threshold as the time gap threshold. The fault onset time threshold determines whether the fault has a direct or indirect impact on the execution unit. Hence, our approach ensures that each execution unit does not contain two contiguous system calls which are separated by a time gap larger than fault onset time threshold.

### 4.3.3 Fault Onset Time Identification

In order to distinguish between faults with a global and local impact, we need to analyze the execution units to identify which threads are affected by the fault and how quickly they are affected. We carefully chose a simple and fast outlier detection algorithm to make PerfCompass practical for online massive system call analysis. We also explored using a clustering algorithm. However we found this yields little accuracy gain with much higher computation overhead. To detect whether a thread is affected by the fault, we first analyze the system call execution time of the execution units in that thread. When calculating system call execution time, it is
important to note that different system call types have different execution time. As a result, it is necessary to distinguish different types of system calls (e.g., `sys_write`, `sys_gettimeofday`) and compute the execution time for each system call type separately.

We search for outlier system call execution time to detect whether the fault affects an execution unit. We use the standard criteria (i.e., > mean + 2 × standard deviation) to detect outliers. If any system call type is identified as an outlier, we infer that this execution unit is affected by the fault.

System call execution time is inherently fluctuating. For example, the execution time of `sys_write` can depend on both the number of bytes to be written as well as any blocking disk activity. These fluctuations can lead to the incorrect detection of spurious outliers. To avoid this, we use a k-length moving average filter to de-noise the system call execution time. Specifically, we compute an average value for a sliding window of k consecutive execution time samples. It is important to set k appropriately as too large of a value will over-smooth the execution time. We found a value of k = 5 works well for all the systems we tested. We have also conducted a sensitivity study on this parameter, which we present in Section 4.4.2

Performance anomalies do not always manifest as changes in system call execution time. For example, if a performance anomaly is caused by an infinite loop, we might observe an increase in the number of system calls generated within the loop when the bug is trigged, compared to the normal execution. As a result, we also include changes in system call frequency as part of the fault impact. We maintain a frequency count for each system call type. When a system call $s_i$ is invoked, we increase the frequency count $C$ for $s_i$ by 1. Next, we compute the time interval $T$ between the start time of the execution unit and the timestamp of $s_i$. We then compute the frequency using $C/T$. We compute frequency in this way because each execution unit represents some individual portion of the application with a certain system call generation rate. Finally, we de-noise and apply outlier detection over the system call frequency in a similar way as we do for the system call execution time.

If a thread consists of multiple execution units, we only consider the first execution unit affected by the fault. We also mark the thread containing the affected execution unit as affected by the fault.

When an execution unit is affected by the fault, we define a fault onset time metric to quantify how fast the fault affects the execution unit and the thread containing it. We compute the fault onset time using the time interval between the start time of the execution unit and the timestamp of the currently processed system call in the first moving average window where either a system call execution time or system call frequency outlier is detected. The rationale for this is we want to identify how long it takes the fault to show any effect on the execution unit.

Figure 4.2 shows a simple example for calculating the fault onset time. Suppose we set the
moving average length \( k = 2 \). The first moving average execution time for \texttt{sys\_write} in the first window is 20 ms. The second moving average execution time of \texttt{sys\_write} in the second window is 19 ms. However, in the third window, the moving average execution time significantly increases to 60 ms. Hence, we infer that this execution unit \( \lambda_1 \) is affected by the fault during the third window. The time elapsed from the start of \( \lambda_1 \) \((t_0)\) to the currently processed system call \((\text{e.g., } \texttt{sys\_write} \text{ in the third window})\) \((t_7)\) is defined as the fault onset time \((\sigma)\). Specifically, the fault onset time \( \sigma_1 \) for \( \lambda_1 \) is \( (t_7-t_0) \).

### 4.3.4 Fault Differentiation Schemes

Our fault differentiation scheme extracts and analyzes two features to determine whether a fault has a global or local impact. We first quantify the fault impact on each thread using the fault onset time to infer whether the thread is affected by the fault directly or indirectly. If the fault onset time of the affected thread is smaller than a pre-defined fault onset time threshold, we say this thread is affected by the fault directly. Otherwise, we say this thread is affected indirectly. We will describe how to set the fault onset time threshold properly in Section 4.3.6.

We then define a \textit{fault impact factor} to compute the percentage of threads affected by the fault directly. If the fault impact factor is close to 100\% (e.g., >90\%), we infer that the fault has a global impact and may be an external fault or internal fault. In this case, we suggest the fault is an external fault because the steps taken to fix an external fault are simple. If the simple fixes are ineffective, the problem is internal. However, if the fault impact factor is significantly less than 100\% (e.g., 80\%), we infer that the fault has a local impact and is thus an internal fault.

We can use the fault impact factor \((\tau)\) alone to correctly localize most faults as either external or internal. However, we found there are borderline cases where it is not clear whether the fault impact was global or local (e.g., \( \tau \in [80\%,90\%] \)). To correctly localize those borderline cases, we compute a \textit{fault onset time dispersion} metric to quantify the differences among the fault onset time values of different affected threads. We use the standard deviation of the fault onset time durations among all the affected threads to compute the fault onset time dispersion. A small fault onset time dispersion means that all threads are affected at roughly the same time, indicating an external fault. In contrast, a large fault onset time dispersion means the threads of the system are affected at different times, indicating an internal fault.

### 4.3.5 Affected System Call Ranking

In addition to distinguishing between external and internal faults, PerfCompass identifies the top affected system call types to provide hints for performance anomaly debugging. For example,
in the Apache external memory cap case, we found an increase in the execution time of \texttt{sys-mmap pgoff}, indicating a memory related problem. In the Hadoop infinite read bug, we found an increase in the frequency of \texttt{sys-read}, a direct result of the bug itself. Knowing which system calls are affected and which system calls are not affected can be helpful for both administrators and developers to understand the root cause of a performance anomaly.

We rank each system call type based on the increase percentage of either the system call execution time or system call frequency. Our ranking algorithm only considers those system calls in the first affected execution unit in each affected thread in order to identify those system calls which are directly affected by the fault. Processing each thread in its entirety runs the risk of identifying system calls affected indirectly by the fault. For example, when a memory cap of a VM has been set too low, it is likely we will see memory related system calls (e.g., \texttt{sys-mmap pgoff}) are affected initially. However, once the machine starts swapping due to lack of memory, we may start to see scheduling-based system calls are more affected (e.g., \texttt{sys-sched-getaffinity}). We use the maximum observed system call execution time percentage increase and system call frequency percentage increase, for each system call type, among all threads, to provide an execution time rank and a frequency rank for each system call type.

Finally, we output both ranked lists in order to provide useful clues for system administrators or software developers to further diagnose external or internal faults. For example, knowing that blocking network I/O based system calls are heavily affected by an external fault could indicate a network contention problem. Alternatively, in the case of an internal fault, knowing blocking network I/O related system calls are affected could help developers localize the buggy segment of code (e.g., portions of the code performing blocking network I/O operations).

### 4.3.6 Fault Localization Enhancement

Unlike CPU and memory related problems, external disk I/O or network faults do not always affect all execution units. For example, disk contention only directly affects those execution units performing disk I/O operations. When the fault impact factor indicates a borderline case (i.e., $\Delta \leq \tau \leq \delta$) and when our system call ranking scheme indicates mainly disk I/O or network related system calls are affected, PerfCompass triggers a filtering mode to achieve more precise fault localization. Figure 4.3 describes our fault localization algorithm, enhanced with the filtering mode. As shown, PerfCompass first filters non-disk I/O and non-network related system calls and then re-computes the fault impact factor $\tau$. We found that the filtering scheme gives more precise diagnosis for external disk I/O or network faults.

During our experiments, we found that the fault onset time threshold for distinguishing between direct and indirect impact varied from application to application. The reason is that the threads of different applications interact with the kernel and between each other in different
Inputs:
α: Fault onset time threshold
β: Fault onset time dispersion threshold
δ: External fault impact factor threshold (90%)
Δ: Internal fault impact factor threshold (80%)

FaultLocalization(α, β)
1. Calculate fault onset time σ for each thread
2. Compute fault impact factor τ using α
3. Rank affected system calls
4. if τ > δ
5. return “external fault”
6. else if τ < Δ
7. return “internal fault”
8. else { /* borderline cases */
9. if top ranked system calls are I/O related
10. Filter non-I/O related system calls
11. Recompute τ
12. if τ > δ
13. return “external fault”
14. else if fault onset time dispersion Ω > β
15. return “internal fault”
16. else
17. return “external fault”
18.}

Figure 4.3: PerfCompass external and internal fault localization algorithm.

ways. Although we found setting a static fault onset threshold (e.g., 0.5 sec) gave correct fault localization results for 23 out of 24 faults we tested, we incorrectly localized one fault. To further improve the accuracy of PerfCompass, we have developed calibration scheme using offline profiling. We run the application under a simple external fault (i.e., an overly low CPU cap) recording the observed fault onset time values. We then use the largest fault onset time value among all the affected threads as the fault onset time threshold for the whole application. We conducted several experiments on calibrating PerfCompass this way using different application workload intensities, mixes, and different application versions. We observe that although the raw calibrated values obtained may slightly vary, the accuracy of our fault localization scheme is not affected by the variations. We can also calibrate the fault onset time dispersion for distinguishing between external and internal faults in a similar way.
4.4 Evaluation

We evaluate PerfCompass using real system performance anomalies caused by different external and internal faults. We first describe our experiment setup followed by the results. Next, we show the overhead imparted by our system.

4.4.1 Experiment Setup

We evaluated PerfCompass using five commonly used server systems: Apache [18], MySQL [79], Tomcat [20], Cassandra [19], and Hadoop [2]. Table 4.2 and 4.3 list all the external and internal faults we tested, respectively. Each of the 17 external faults represents a common multi-tenancy or environment issue such as interference from other co-located applications, insufficient resource allocation, or network packet loss. We also tested 7 internal faults which are real software bugs found in real world bug repositories (e.g., Bugzilla) by searching for performance related terms such as slowdown, hangs, and 100% CPU usage. We then follow the instructions given in the bug reports to reproduce the bugs.

Six of these seven internal software bugs are bugs which cause the system to hang and one bug causes a degradation in performance. We choose these bugs to evaluate PerfCompass in order to demonstrate that our tool is capable of handling bugs causing both a slowdown and system hang. We choose more bugs causing the system to hang because recent work [53, 129] has identified that those bugs are both difficult to debug and cause of a significant portion of problems in the cloud.

We use Apache 2.2.22, MySQL 5.5.29, Tomcat 7.0.28, Cassandra 1.2.0-beta, and Hadoop 2.0.0-alpha for evaluating the external faults of those systems. For each internal fault, we used the version specified in the bug report. For all faults, PerfCompass is triggered when a performance anomaly is detected (e.g., response time > 100 ms, progress score is 0), using the past 1.5 minutes of collected system call trace data for analysis. The accuracy of PerfCompass is not affected by the length of the history data as long as the data cover the fault manifestation. Our experiments show that 1.5 minutes is sufficient for all the faults we tested.

In order to evaluate PerfCompass under dynamic workloads with realistic time variations, we use the following workloads during our experiments.

- **Apache** - We employ a benchmark tool [6] to send requests to the Apache server at different intensity levels following the per-minute workload intensity observed in a NASA web server trace starting at 1995-07-01:00:00 [14].

- **MySQL** - We use a open source MySQL benchmark tool called “Sysbench” [13]. We use the *oltp* test in *complex* mode.
- **Tomcat** - We randomly request different example servlets and JSPs included in Tomcat following the same workload intensity observed in the NASA web server trace [14] as Apache.

- **Cassandra** - We use a simple workload which creates a table and inserts various entries into the table.

- **Hadoop** - We use the Pi calculation sample application with 16 map and 16 reduce tasks.

PerfCompass has six parameters which need to be configured in order to achieve optimal results. They are the fault onset time threshold ($\alpha$), the fault onset time dispersion threshold ($\beta$), the window size for calculating the moving average ($k$), the threshold for outlier detection ($\Theta$), external fault impact factor threshold ($\delta$), and the internal fault impact factor threshold ($\Delta$). $k$, $\Theta$, $\delta$, and $\Delta$ are not sensitive and as a result, we use the same settings for different applications. We found that different $\alpha$ and $\beta$ values are needed for each application to obtain optimal results.

During our experiments, the default parameter setting in PerfCompass is as follows: 1) $k$ is 5; 2) $\Theta$ is mean + (2 × standard deviation); and 3) $\delta$ and $\Delta$ for differentiating external and internal faults are 90% and 80%, respectively. These settings are configurable and we have not attempted to calibrate or tune them to any system. We found those default settings work well for all the systems and faults we tested.

We obtained calibrated $\alpha$ and $\beta$ values for each system using a simple external CPU cap fault (i.e., setting an overly low CPU cap) with different workload intensity levels from those used in the fault localization experiments. We intentionally configure this way in order to evaluate whether our calibration requires the same workload as the production-run failure. Our results show that changing the workload intensity does not affect our fault localization results. The derived $\alpha$ are 60 ms for Apache and Tomcat, 300 ms for MySQL, and 400 ms for Cassandra and Hadoop. The calibrated $\beta$ values are 7 ms for Apache, 17 ms for MySQL, 5 ms for Tomcat, 28 ms for Cassandra, and 39 ms for Hadoop.

We repeat each fault injection three times and report the mean and standard deviation of the fault impact factor values over all 3 runs.

We conducted our experiments on two different virtualized clusters. The Apache and MySQL experiments were conducted on a cluster where each host has a dual-core Xeon 3.0GHz CPU and 4GB memory, and runs 64bit CentOS 5.2 with Xen 3.0.3. The Tomcat, Cassandra, and Hadoop experiments were conducted on a cluster where each node is equipped with a quad-

---

1We used a static 1 second time segmentation gap threshold to segment each trace during calibration. We also tested other static time segmentation gap threshold values (e.g., 0.5 sec, 1.5 sec) and found it did not affect our calibration results.
core Xeon 2.53Ghz CPU along with 8GB memory with KVM 1.0. In both cases, each trace was collected on a guest VM using LTTng 2.0.1 running 32-bit Ubuntu 12.04 kernel version 3.2.0.

4.4.2 Online Fault Localization Results

We now present our online fault localization and inference results. We begin by presenting our external and internal fault differentiation results. We then present our top affected system call ranking results followed by a sensitivity study on the key parameters of our system.

Fault Differentiation Results

Figure 4.4 and Figure 4.5 show the fault impact factor and fault onset time dispersion results using the calibrated fault onset time threshold values, respectively. As shown in Figure 4.4, the impact factors of all the external faults are greater than 90%, indicating them as external with no further analysis required. Similarly, 4 out of 7 of the internal faults are below the 80%, correctly indicating each case as internal. There are three borderline cases which require to use our fault onset time dispersion metric. Fault 15 is an internal Tomcat bug with an impact factor of 83%. Figure 4.5 shows the fault onset time dispersion values for the borderline cases we tested, normalized using the calibrated fault onset time dispersion threshold values. We observe that the fault onset time dispersion value of the internal case is significantly larger than the...
fault onset time dispersion threshold (i.e., $2 \times$) for Tomcat, indicating it is an internal fault. Faults 23 and 24 have fault impact factors of about 80%, which makes them borderline cases as well. As shown in Figure 4.5, the fault onset time dispersion values for the two internal Hadoop cases are significantly higher than the fault onset time dispersion threshold (i.e., $2.3 \times, 2.2 \times$) for Hadoop, correctly indicating both of them as internal bugs.

We also evaluated PerfCompass using a static 0.5 second fault onset time threshold as shown by Figure 4.6 and Figure 4.7. We observe that using a static fault onset time value, PerfCompass can still correctly localize 23 out of the 24 tested faults. As shown in Figure 4.7, we incorrectly identify fault 15 as external since its fault onset time dispersion value is lower than the calibrated fault onset time dispersion threshold using a simple external CPU cap fault.

Setting a fault impact factor threshold of 90% would have allowed PerfCompass to correctly localize each fault as either external or internal without using our fault impact factor dispersion metric, thus reducing the overhead of PerfCompass. However, setting a relatively high fault impact factor threshold (e.g., 90%) would likely over-fit our scheme to our experimental data. This may increase the probability that PerfCompass would incorrectly localize an external fault as an internal fault for systems we have not tested. In addition, we want to avoid setting a hard threshold for distinguishing between internal and external faults (e.g., 90% being external and 89% being internal). As a result, we decided to use a conservative fault impact factor threshold and employ a hybrid two tier scheme to better capture the multi-dimensional difference between

---

**Figure 4.5:** The normalized fault onset time dispersion ($\Omega$) values using calibrated fault onset time threshold ($\alpha$).
external and internal faults.

System Call Ranking Results

Tables 4.4 and 4.5 show the top three affected system calls identified by our ranking scheme for the external faults for C++ and Java systems, respectively. Tables 4.6 shows the top three affected system calls identified by our ranking scheme for the internal faults. We consider both execution time and frequency changes while identifying top affected system calls. For example, “Time:sys_write (1)” means that `sys_write` is the highest ranked system call whose execution time increase is the largest. The last column in each table shows our inference results. For the external faults, this column describes how the affected system calls indicate which subsystems are most affected by the faults. For the internal faults, our system call ranking results give hints that may help developers localize the buggy function. Note that PerfCompass is designed to be a light-weight online fault inference tool that focuses on providing useful clues for diagnosing both external and internal faults. Thus, we do not intend to use PerfCompass to pinpoint the exact root cause of a performance anomaly (e.g., buggy function name). We first discuss a subset external faults followed by a subset of internal faults.

Fault 8: This external fault is due to an incorrectly set external memory cap. The indicator of this fault is the `sys_fsync` system call. MySQL tries to keep as much of the database in physical memory as possible, flushing to disk only when necessary. The low amount of physical
memory causes the system to use the disk more frequently, which can require additional memory to disk synchronization using the `sys_fsync` system call.

Fault 18: This is an external fault due to disk contention. We can see the execution time of `sys_futex` and `sys_write` are affected, indicating a blocking I/O problem. In turn, this indicates external disk contention.

Fault 21: This is an external fault due to packet loss. The increase in the execution time of `sys_futex` and `sys_socketcall` indicates a blocking network I/O problem. The execution time increase is a result of the dropped packets being re-transmitted.

Fault 5: Fault 5 is an internal CPU hog bug that occurs as a result of an infinite loop in which Apache attempts to make a blocking call on a socket. However, a flag preventing it from successfully making the call has not been cleared. The return value of the call is not checked correctly resulting in Apache re-trying the blocking call continuously. We found an increase in the frequency and execution time of the `sys_ipc` system call, which is a result of Apache repeatedly attempting to make the blocking call.

Fault 10: This internal fault is a deadlock bug caused by attempting to execute the `INSERT_DELAYED` statement on a locked table. The thread attempting to execute the `INSERT_DELAYED` statement tries to continuously access the locked table. We found an increase in the execution time of the `sys_select` and `sys_futex` system calls. `sys_select` is used to

Figure 4.7: The normalized fault onset time dispersion ($\Omega$) values using a 0.5 second static fault onset time threshold ($\alpha$).
allow a program to monitor multiple file descriptors. Its execution time increase is due to the
deadlocked function attempting to perform an I/O operation. Similarly, the execution time
increase we observe with the *sys_futex* system call is a result of system trying unsuccessfully
to acquire a lock on the already locked table. We also see an increase in the frequency of the
*sys_stat64* system call, which is commonly used to first ensure a file exists before performing
a requested I/O operation. This frequency increase is due to the repeated I/O attempts.

**Fault 15:** This is an internal fault caused by an incorrectly updated atomic counter value
that is used to keep track of how many requests are being processed. This causes all other threads
to hang indefinitely as the system incorrectly believes it cannot handle any more requests. Here
we see an increase in both execution time and frequency of the *sys_futex* system call. This
indicates a shared variable is being checked repeatedly.

**Fault 23:** This is an internal bug with HDFS where the system does not correctly check
a header field. This causes HDFS to continuously attempt to read from a socket, waiting for
the content that does not exist. This process continues until the job times out. Here, each of
system calls affected is a result of the bug. The frequency increase of the I/O system calls,
*sys_close*, *sys_read*, and *sys_stat64* are results of an infinite loop continuously attempting
to read. We also see an increase in the execution time of the *sys_futex*, *sys_socketcall*, and
*sys_epoll_wait* system calls as results of the read operations in the infinite loop.

**Filtering Mode**

We now show the calibrated impact factor values with and without filtering mode enabled in
Figure 4.8. The cases shown are external cases with highly ranked I/O related system calls.
As shown, PerfCompass is still able to correctly quantify the global vs. local impact of most
faults without filtering mode enabled. However, case 17 can be considered a borderline case as
a result of the error bars of the average impact factor. When we apply filtering mode to that
case, however, we find the impact factor increases substantially from 91+/-6.7% to 99+/-0.6%.
Although this is the only case which shows a significant benefit from filtering mode, we believe
this mode can be useful in allowing developers more flexibility when defining the fault impact
factor threshold value.

**Sensitivity Study**

We conducted a sensitivity study in order to determine how different configuration parameters
affect PerfCompass. The parameters we tested were the static fault onset time threshold and
the sliding window size. We use the calibrated fault onset time threshold values to test the
sliding window size parameter. The results of our study are shown in Table 4.7. As shown, the
fault localization results of PerfCompass are not sensitive to different parameter settings when
Figure 4.8: The filtering mode effect using calibrated fault onset time thresholds for different faults where the filtering mode is triggered.
they are set to reasonable values. Specifically, with the variations shown, we correctly localize 23 out of the 24 tested faults as external or internal without calibration. With calibration, we correctly localize all 24 tested faults as external or internal.

4.4.3 PerfCompass Overhead

We evaluated the overhead imposed by PerfCompass during normal application execution. For Apache, Tomcat, and MySQL we used httperf to send a fixed number of requests to one of our benchmark systems, recording the average response time both with and without PerfCompass. We used a request rate of 100 requests per second for Apache, 50 requests per second for Tomcat, and 20 requests per second for MySQL. For Hadoop we ran the Pi example job, recording the average processing time both with and without PerfCompass. For Cassandra, we ran a simple database insertion workload, recording the average processing time both with and without PerfCompass. We ran all tests 5 times, reporting the mean and standard deviation. Figure 4.9 shows the runtime collection overhead imposed by PerfCompass on each of the tested systems. As the figure shows, we impart an average overhead of 2.1% runtime overhead to the server. Specifically, we impart 0.81% to Apache, 2.9% to Hadoop, 2.2% to MySQL, 1.3% to Tomcat, and 3.3% to Cassandra.

We now evaluate the online fault localization and inference time of PerfCompass. Note that the fault analysis is only triggered after a performance anomaly is detected. Table 4.8 shows the total fault localization and inference time for all the faults we tested. We ran each experiment 5 times reporting the mean and standard deviation for each case. As the table shows, our analysis algorithms take tens of seconds to complete for most cases and with an upper bound of 97 seconds. Thus, our tool is practical for online fault localization and inference.

We also measured the resource usage overhead of PerfCompass. We found PerfCompass imparts between 2-3% CPU load and has a small memory footprint (about 256KB). Thus, we
believe that PerfCompass is light-weight and practical for online production system diagnosis.

4.5 Summary

In this chapter, we have presented PerfCompass, a novel online performance anomaly fault localization and inference tool for production cloud infrastructures. PerfCompass can efficiently distinguish between external and internal faults without requiring source code or runtime instrumentation. PerfCompass extracts fine-grained fault features from kernel-level system call traces and uses those fault features to perform online fault localization. PerfCompass can also provide useful hints about both external and internal faults by identifying top affected system calls. By focusing on kernel-level system call events, PerfCompass can be applied to different programs written in C/C++ or Java. We have implemented PerfCompass and evaluated it using a variety of commonly used open source server systems including Apache, MySQL, Tomcat, Cassandra, and Hadoop. We tested PerfCompass using 24 real cloud faults consisting of common environment issues in the shared cloud infrastructure and real performance bugs. Our results show that our fault differentiation scheme can achieve 100% fault localization accuracy and provide useful fault hints. PerfCompass is light-weight, which makes it practical for use in production cloud infrastructures.
Table 4.2: Descriptions of the 17 external faults we tested.

<table>
<thead>
<tr>
<th>System name</th>
<th>Fault ID</th>
<th>Fault description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>1</td>
<td>CPU cap problem: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Memory cap problem: improperly setting the VM's memory cap to too low causes insufficient memory allocation.</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>I/O interference problem: a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Packet loss problem: using the tc command causes the network to randomly drop 10% of the packets.</td>
</tr>
<tr>
<td>MySQL</td>
<td>6</td>
<td>I/O interference problem: a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>CPU cap problem: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Memory cap problem: improperly setting the VM's memory cap to too low causes insufficient memory allocation.</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Packet loss problem: using the tc command causes the network to randomly drop 10% of the packets.</td>
</tr>
<tr>
<td>Tomcat</td>
<td>12</td>
<td>Packet loss problem: using the tc command causes the network to randomly drop 10% of the packets.</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>CPU cap problem: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>Memory cap problem: improperly setting the VM's memory cap to too low causes insufficient memory allocation.</td>
</tr>
<tr>
<td>Cassandra</td>
<td>16</td>
<td>CPU cap problem: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>Packet loss problem: using the tc command causes the network to randomly drop 10% of the packets.</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>I/O interference problem: a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.</td>
</tr>
<tr>
<td>Hadoop</td>
<td>20</td>
<td>CPU cap problem: improperly setting the VM's CPU cap to too low causes insufficient CPU allocation.</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>Packet loss problem: using the tc command causes the network to randomly drop 10% of the packets.</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>I/O interference problem: a co-located VM causes a disk contention interference problem by running a disk intensive Hadoop job.</td>
</tr>
</tbody>
</table>
Table 4.3: Descriptions of the 7 internal faults we tested.

<table>
<thead>
<tr>
<th>System name</th>
<th>Fault ID</th>
<th>Fault description (type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>5</td>
<td><strong>Flag setting bug (hang)</strong>: deleting a port Apache is listening to and then restarting the server causes Apache to attempt to make a blocking call on a socket when a flag preventing blocking calls has not been cleared. The code does not check for this condition and continuously re-tries the call (#37680).</td>
</tr>
<tr>
<td>MySQL</td>
<td>10</td>
<td><strong>Deadlock bug (hang)</strong>: a MySQL deadlock bug that occurs when each of the two connections locks one table and tries to lock the other table. If one connection tries to execute a <code>INSERT DELAYED</code> command on the other while the other is sleeping, the system will become deadlocked (#54332).</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td><strong>Data flushing bug (slowdown)</strong>: truncating a table causes a 5x slowdown in table insertions due to a bug with the InnoDB storage engine for big datasets. InnoDB fails to mark truncated data as deleted and constantly allocates new blocks (#65615).</td>
</tr>
<tr>
<td>Tomcat</td>
<td>15</td>
<td><strong>Infinite wait bug (hang)</strong>: A counter value is not updated correctly causing the request processing threads of Tomcat to hang (#53173).</td>
</tr>
<tr>
<td>Cassandra</td>
<td>19</td>
<td><strong>Infinite loop bug (hang)</strong>: trying to alter a table when the table includes collections causes it to hang, consuming 100% CPU due to an internal problem with the way Cassandra locks tables for updating (#5064).</td>
</tr>
<tr>
<td>Hadoop</td>
<td>23</td>
<td><strong>Infinite read bug (hang)</strong>: HDFS does not check for an overflow of an internal content length field causing HDFS transfers larger than 2GB to try to repeatedly read from an input stream until job time out (#HDFS-3318).</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td><strong>Thread shutdown bug (hang)</strong>: when the AppLogAggregator thread dies unexpectedly (e.g. due to a crash), the task waits for an atomic variable to be set indicating thread shutdown is complete. As the thread has died already, the variable will never be set and the job will hang indefinitely (# MAPREDUCE-3738).</td>
</tr>
</tbody>
</table>
Table 4.4: System call ranking results for the external faults we tested for C/C++ systems.

<table>
<thead>
<tr>
<th>System Name</th>
<th>Fault ID</th>
<th>Top affected system calls (rank)</th>
<th>Fault hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>1</td>
<td><code>Time:sys_waitpid(1), sys_mmap_pgooff(2), sys_socketcall(3)</code></td>
<td>Execution time increase of <code>sys_waitpid</code> indicates a CPU contention problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Frequency:None</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td><code>Time:sys_clone(1), sys_mmap_pgooff(2), sys_socketcall(3)</code></td>
<td>Execution time increase of <code>sys_mmap_pgooff</code> and <code>sys_clone</code> indicates a memory allocation problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Frequency:None</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td><code>Time:sys_select(1), sys_fcntl64(2), sys_socketcall(3)</code></td>
<td>Increase in frequency of <code>sys_fcntl64</code> along with increase in execution time of <code>sys_select</code> indicates a disk contention problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Frequency:sys_fcntl64(1)</code></td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td>4</td>
<td><code>Time:sys_socketcall(1), sys_open(2), sys_close(3)</code></td>
<td><code>sys_socketcall</code> is used for network I/O. <code>sys_open</code> and <code>sys_close</code> also related to sending data over the network. This indicates a non-blocking network I/O problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Frequency:None</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td><code>Time:sys_futex(1), sys_read(2)</code></td>
<td>Increase in execution time of <code>sys_futex</code> and <code>sys_read</code> indicates blocking I/O problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Frequency:None</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td><code>Time:sys_futex(1)</code></td>
<td><code>sys_futex</code> affected by itself indicates issue related to locking. Indicates possible CPU contention or memory problem. Lack of any other system calls being affected indicates a CPU contention problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Frequency:None</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td><code>Time:sys_futex(1), sys_fsync(2), sys_socketcall(3)</code></td>
<td><code>sys_fsync</code> used to flushes buffer cache data to disk, which can occur when low on memory. Indicates a memory related problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Frequency:None</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td><code>Time:sys_futex(1), sys_socketcall(2)</code></td>
<td>Increase in execution time of <code>sys_futex</code> and <code>sys_socketcall</code> indicates a blocking network I/O problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Frequency:None</code></td>
<td></td>
</tr>
</tbody>
</table>
Table 4.5: System call ranking results for the external faults we tested for Java systems.

<table>
<thead>
<tr>
<th>System Name</th>
<th>Fault ID</th>
<th>Top affected system calls (rank)</th>
<th>Fault hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomcat</td>
<td>12</td>
<td>Time:sys_futex(1),sys_socketcall(2), sys_stat64(1),sys_futex(2)</td>
<td>sys_socketcall and sys_futex execution time increase indicates a blocking network I/O problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frequency:sys_stat64(1),sys_futex(2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>Time:sys_futex(1),sys_sched_getaffinity(2),sys_write(3),sys_futex(1)</td>
<td>Execution time and frequency increase of sys_futex along with execution time increase of sys_sched_getaffinity indicates a CPU contention problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frequency:sys_futex(1)</td>
<td>Increase in frequency of sys_sched_yield indicates increase in frequency of context switching, indicates a memory problem.</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>Time:sys_futex(1),sys_socketcall(2),sys_stat64(3),sys_sched_yield(1)</td>
<td>Increase in execution time of sys_socketcall and sys_futex indicates a CPU contention problem.</td>
</tr>
<tr>
<td>Cassandra</td>
<td>16</td>
<td>Time:sys_futex(1),sys_poll(2),sys_socketcall(3),sys_close(1),sys_unlink(2),sys_futex(3)</td>
<td>Increase in execution time of sys_futex and sys_poll indicates a CPU contention problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frequency:sys_close(1),sys_unlink(2),sys_futex(3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>Time:sys_futex(1),sys_socketcall(2),sys_sched_getaffinity(3),sys_close(1),sys_futex(2),sys_sched_yield(3)</td>
<td>Increase in execution time of sys_socketcall and sys_futex indicates a blocking network I/O problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frequency:sys_close(1),sys_futex(2),sys_sched_yield(3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>Time:sys_futex(1),sys_poll(2),sys_write(3),sys_close(1),sys_futex(2)</td>
<td>Increase in execution time of sys_futex and sys_write indicates a blocking I/O problem. This indicates a disk contention problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frequency:sys_close(1),sys_futex(2)</td>
<td></td>
</tr>
<tr>
<td>Hadoop</td>
<td>20</td>
<td>Time:sys_futex(1),sys_socketcall(2),sys_epoll_wait(3),sys_close(1),sys_open(2)</td>
<td>Increase in execution time of sys_epoll_wait and sys_futex indicates a CPU contention problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frequency:sys_close(1),sys_open(2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>Time:sys_futex(1),sys_socketcall(2),sys_read(3),sys_close(1),sys_read(2)</td>
<td>sys_futex and sys_socketcall execution time increase indicates a blocking network I/O problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frequency:sys_close(1),sys_read(2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>Time:sys_futex(1),sys_epoll_wait(2),sys_read(3),sys_close(1),sys_read(2)</td>
<td>Increase in execution time and frequency of sys_read indicates a disk contention problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frequency:sys_close(1),sys_read(2)</td>
<td></td>
</tr>
</tbody>
</table>
Table 4.6: System call ranking results for the internal faults we tested.

<table>
<thead>
<tr>
<th>System Name</th>
<th>Fault ID</th>
<th>Top affected system calls (rank)</th>
<th>Fault hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>5</td>
<td><strong>Time:</strong> sys_ipc(1), sys_close(2), sys_waitpid(3) <strong>Frequency:</strong> sys_ipc(1), sys_open(2)</td>
<td>Frequency increase of <code>sys_ipc</code> indicates significant increase in inter-thread signalling. <strong>Hints:</strong> blocking operation (e.g., blocking socket call) in an infinite loop.</td>
</tr>
<tr>
<td>MySQL</td>
<td>10</td>
<td><strong>Time:</strong> sys_select(1), sys_futex(2), sys_fsync(3) <strong>Frequency:</strong> sys_stat64(1), sys_open(2), sys_read(3)</td>
<td>Increase in execution time of <code>sys_futex</code> and frequency of I/O related system calls indicate repeated attempts to acquire lock and perform I/O. <strong>Hints:</strong> possible deadlocked I/O functions.</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td><strong>Time:</strong> sys_futex(1), sys_select(2), sys_socketcall(3) <strong>Frequency:</strong> None</td>
<td>Execution time increase of <code>sys_select</code> and <code>sys_futex</code> indicate increase of writing and reading to disk. <strong>Hints:</strong> possible problem with data flushing functions.</td>
</tr>
<tr>
<td>Tomcat</td>
<td>15</td>
<td><strong>Time:</strong> sys_futex(1), sys_close(2), sys_pipe(3) <strong>Frequency:</strong> sys_close(1), sys_open(2), sys_futex(3)</td>
<td>Increase in execution time and frequency of <code>sys_futex</code> indicates issue with locking operations. <strong>Hints:</strong> shared variable checking functions.</td>
</tr>
<tr>
<td>Cassandra</td>
<td>19</td>
<td><strong>Time:</strong> sys_futex(1), sys_fstat64(2), sys_stat64(3) <strong>Frequency:</strong> sys_close(1), sys_lstat64(2), sys_unlink(3)</td>
<td>Increase in execution time of <code>sys_futex</code>, <code>sys_fstat64</code>, and <code>sys_stat64</code> along with frequency increase of <code>sys_close</code> indicates significant increase in I/O operations. <strong>Hints:</strong> I/O functions using locking operations in an infinite loop.</td>
</tr>
<tr>
<td>Hadoop</td>
<td>23</td>
<td><strong>Time:</strong> sys_futex(1), sys_socketcall(2), sys_epoll_wait(3) <strong>Frequency:</strong> sys_close(1), sys_read(2), sys_stat64(3)</td>
<td>Increase in execution time of <code>sys_futex</code>, <code>sys_socketcall</code>, and <code>sys_epoll_wait</code>, along with increase in frequency of <code>sys_close</code>, <code>sys_read</code>, and <code>sys_stat64</code> indicates issue with network I/O operation. <strong>Hints:</strong> I/O operation involving functions reading over the network.</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td><strong>Time:</strong> sys_futex(1), sys_socketcall(2), sys_epoll_wait(3) <strong>Frequency:</strong> sys_close(1), sys_stat64(2), sys_rt_sigaction(3)</td>
<td>Increase of <code>sys_futex</code> and <code>sys_epoll_wait</code> indicates issue with locking operations. <strong>Hints:</strong> shared variable checking functions.</td>
</tr>
</tbody>
</table>
Table 4.7: The fault impact factor ($\tau$) results under different parameter settings of the PerfCompass system. The numbers in bold represent the default settings we used.

<table>
<thead>
<tr>
<th>System name</th>
<th>Fault ID</th>
<th>Internal or External</th>
<th>Static fault onset threshold</th>
<th>Sliding window size (k)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.5 sec</td>
<td>1 sec</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Apache</td>
<td>1</td>
<td>External</td>
<td>100%</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>External</td>
<td>99%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>External</td>
<td>98%</td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>External</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Internal</td>
<td>62%</td>
<td>62%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>62%</td>
</tr>
<tr>
<td>MySQL</td>
<td>6</td>
<td>External</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>External</td>
<td>100%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>External</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>External</td>
<td>100%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Internal</td>
<td>41%</td>
<td>41%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>42%</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Internal</td>
<td>68%</td>
<td>72%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69%</td>
</tr>
<tr>
<td>Tomcat</td>
<td>12</td>
<td>External</td>
<td>94%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>External</td>
<td>100%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>External</td>
<td>96%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>Internal</td>
<td>83%</td>
<td>83%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89%</td>
</tr>
<tr>
<td>Cassandra</td>
<td>16</td>
<td>External</td>
<td>95%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>External</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>External</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>Internal</td>
<td>46%</td>
<td>46%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>47%</td>
</tr>
<tr>
<td>Hadoop</td>
<td>20</td>
<td>External</td>
<td>97%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>External</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>External</td>
<td>98%</td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>Internal</td>
<td>79%</td>
<td>79%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>81%</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>Internal</td>
<td>78%</td>
<td>78%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>80%</td>
</tr>
</tbody>
</table>
Table 4.8: PerfCompass analysis time for each fault we tested.

<table>
<thead>
<tr>
<th>Fault Name</th>
<th>Fault ID</th>
<th>Fault localization and inference time</th>
<th>Num. of System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>1</td>
<td>7 ± 1 sec</td>
<td>0.3 ± 0.83 million</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1 ± 1 sec</td>
<td>0.1 ± 0.2 million</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4 ± 1 sec</td>
<td>1.1 ± 0.1 million</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9 ± 1 sec</td>
<td>1.1 ± 0.1 million</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>26 ± 4 sec</td>
<td>1.1 ± 0.1 million</td>
</tr>
<tr>
<td>MySQL</td>
<td>6</td>
<td>38 ± 2 sec</td>
<td>1.6 ± 0.1 million</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>42 ± 2 sec</td>
<td>1.9 ± 0.003 million</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>54 ± 7 sec</td>
<td>1.9 ± 0.003 million</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>61 ± 5 sec</td>
<td>1.9 ± 0.003 million</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>22 ± 0.1 sec</td>
<td>1.9 ± 0.003 million</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>7 ± 0.1 sec</td>
<td>1.9 ± 0.003 million</td>
</tr>
<tr>
<td>Tomcat</td>
<td>12</td>
<td>18 ± 0.7 sec</td>
<td>0.4 ± 0.02 million</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>11 ± 0.3 sec</td>
<td>0.3 ± 0.01 million</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>29 ± 0.8 sec</td>
<td>0.8 ± 0.02 million</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>14 ± 0.9 sec</td>
<td>0.4 ± 0.03 million</td>
</tr>
<tr>
<td>Cassandra</td>
<td>16</td>
<td>2 ± 0.1 sec</td>
<td>0.6 ± 0.01 million</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>2 ± 0.1 sec</td>
<td>1.2 ± 0.007 million</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>2 ± 0.1 sec</td>
<td>1.1 ± 0.03 million</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>38 ± 1 sec</td>
<td>1.6 ± 0.03 million</td>
</tr>
<tr>
<td>Hadoop</td>
<td>20</td>
<td>37 ± 2 sec</td>
<td>1.6 ± 0.02 million</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>97 ± 3 sec</td>
<td>4 ± 0.1 million</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>65 ± 2 sec</td>
<td>2.6 ± 0.03 million</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>38 ± 2 sec</td>
<td>1.5 ± 0.09 million</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>34 ± 1 sec</td>
<td>1.4 ± 0.04 million</td>
</tr>
</tbody>
</table>
Chapter 5

PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures

In this chapter we discuss the third component of our framework, PerfScope. When a fine-grained localization system (e.g., PerfCompass) identifies that a performance anomaly is caused by an internal software bug, PerfScope is triggered to provide feedback to developers to help them diagnose the problem. We begin with a brief introduction of our work, followed by a detailed discussion of PerfScope.

5.1 Introduction

It is a notoriously difficult task to diagnose a performance bug [58,81,101] that occurred inside a production cloud computing infrastructure\(^1\). The reasons are multifold. First, it is difficult, if not totally impossible, to reproduce a production-run performance bug outside the cloud infrastructure because those performance bugs often only manifest under specific conditions (e.g., particular user inputs, certain system configurations, non-deterministic system events). Thus, it is hard to apply existing offline debugging tools such as GDB directly to diagnose those performance bugs that cannot be reproduced outside the production cloud infrastructure.

Second, unlike crash failures, performance bugs often provide little diagnostic information. Many performance bugs even do not produce any error message since they are unexpected by

\(^1\)We do not distinguish different types of cloud systems (e.g., infrastructure-as-a-service clouds, platform-as-a-service clouds) since we believe that our problem and approach apply to different cloud systems.
the developer. To exacerbate the problem, it is often impractical to perform detailed system execution tracing inside the cloud infrastructure due to concerns about prohibitive cost and user privacy. However, a modern server system typically consists of tens of thousands of functions. Searching the buggy functions among tens of thousands of functions without any clue becomes an impossible task in many cases.

In this chapter, we propose PerfScope, a practical online bug inference tool to help the application developer understand why a performance bug occurs in the production cloud computing infrastructure. One big advantage of PerfScope over existing offline debugging tools is that PerfScope does not require production-run bug reproduction by achieving online bug inference. PerfScope also does not require application source code or any system instrumentation, which makes it practical for production cloud infrastructures. PerfScope is application-agnostic, which can be applied to any program (e.g., C/C++, Java, Python, etc.) running inside the cloud system. Our work focuses on diagnosing performance bugs [58] that are defined as software bugs causing performance anomalies (e.g., software hang, performance slowdown). Performance anomalies caused by other problems (e.g., hardware faults, configuration issues, kernel bugs) are outside the scope of this work.

When a performance anomaly is detected by an existing online anomaly detection tool [40,

---

**Figure 5.1:** A subset of function call graph for a real HDFS performance bug. When an input file larger than 2GB is used, the internal variable (int) representing the content length of the input file overflows. This causes the call to `in.read(buf)` at line 81 to never return -1 causing an infinite loop. The bug point is highlighted in bold.

```java
public static void copyBytes(...) 
...
76  while (bytesRead >= 0){
    ....
81   bytesRead = in.read(buf);
  }
...
```

```java
  public int read(...) 
  153  return reader.doIO(...);
```

```java
  int doIO(...) 
  142  return performIO(buf);
```

```java
  int performIO(...) 
  54   return channel.read(buf);
```

Bug related function

---

---

---
PerfScope is triggered to perform online bug inference by analyzing a window (e.g., five minutes) of recent system call traces collected on the faulty server pinpointed by a component-level fault localization tool [34, 83]. The rationale behind our approach is twofold. First, a system call trace can be easily collected via kernel tracing tool\(^2\) such as Linux Tracing Tool next generation (LTTng) [43] in production cloud computing infrastructures with low overhead. Moreover, compared to other low-level tracing tools (e.g., hardware performance counter monitoring tool [92]), the system call tracing tool works in virtualized environments, which makes it suitable for cloud infrastructures. Second, we observe that performance anomalies are often caused by anomalous interactions between the application and the kernel. Since applications use system calls to interact with the kernel, we can detect anomalous application-kernel interactions by closely monitoring system calls. For example, Figure 5.1 shows a real performance bug in the HDFS system: an overflow issue causes the \texttt{while} loop to never end. This buggy execution produced 1.4 million system calls. As we will show in Section 5.3, we find that over 90% buggy functions produce system calls by performing static analysis over 228 real performance bugs.

However, it is a challenging task to identify bug-related functions from the massive, noisy raw system call traces. First, we need to develop efficient online system call trace analysis algorithms that can quickly identify the anomalous system call sequences from millions of system calls. Second, we need to develop fast and robust algorithms that can accurately map those anomalous system call sequences to specific bug-related application functions. Since our goal is to support different applications (e.g., both compiled and interpreted programs), we cannot rely on the program counter information to trace back to the application functions, which does not work for interpreted programs and some compiled programs using certain compilation optimizations.

To address those challenges, PerfScope first uses light-weight unsupervised learning techniques to identify anomalous system call sequences. Since the system call trace for a multithreaded server program is often huge, we first segment the raw system call trace into fine-grained units of closely related system calls called execution units. We then characterize different execution units using various features (e.g., system call appearance vector, system call execution time vector, system call frequency vector). Next, we use a hybrid anomaly detection scheme (i.e., hierarchical clustering plus outlier detection) to identify abnormal execution units.

PerfScope then uses those abnormal execution units as clues to identify the bug-related functions. We propose a signature-driven approach that creates a robust signature for each application function offline outside the production computing environment. Each function signature maps the function into a set of closed frequent system call episodes (e.g., \texttt{sys\_write}, \texttt{sys\_read}). When a production-run performance bug occurs, we extract the closed frequent system call episodes from those anomalous execution units and map those episodes back to a

\(^2\)Unlike the user level system call tracing tools such as ptrace [9], our experiments show that the kernel-based system call tracing only imposes on average 1.8% overhead to the tested server applications.
short list of candidate buggy functions using the function signatures. We also provide a ranking to those identified functions based on the abnormality degree observed in those frequent system call episodes. Note that we do not view PerfScope as the final debugging tool. Instead, PerfScope provides the developer with a small number of candidate buggy functions to examine, which makes production-run performance bug diagnosis much more tractable.

Specifically, this chapter makes the following contributions:

- We propose an online performance bug inference tool that works for production cloud computing environments without requiring offline bug reproduction, application source code, or runtime system instrumentation.

- We present a robust application function signature extraction approach that works for different types of applications (e.g., both compiled and interpreted programs) running inside cloud computing environments.

- We describe a set of online system call trace analysis algorithms that can quickly process millions of system calls to produce a short list of bug-related functions.

- We evaluated PerfScope using real performance bugs in popular open source systems (Hadoop, HDFS, Cassandra, Tomcat, Apache, Lighttpd, MySQL). The results show that PerfScope is effective, which can narrow down the search scope for the bug-related functions to a small percentage (0.03-2.3%). The real bug-related functions are ranked within top five candidates in 9 out of 12 cases and are always ranked within top 12. PerfScope only imposes on average 1.8% runtime overhead to the tested server applications.

**Assumptions:** We design PerfScope based on the following assumptions: 1) Performance bugs manifest as time or frequency changes in system calls. As mentioned earlier, over 90% of the real performance bugs we studied generate system calls; 2) We do not have access to application source code or any other application knowledge; and 3) We assume that function signatures can be profiled offline outside the production environments and remain stable under different inputs, workload intensities, and resource allocation settings. Although we cannot assure that this assumption holds on every system, our experiments with seven popular server systems under different workload conditions show that our approach is applicable to different types of functions without requiring application specific tuning.

The rest of the chapter is organized as follows. Section 5.2 describes the design of the PerfScope system. Section 5.3 presents our experimental evaluation. Finally, the chapter concludes in Section 5.4.
5.2 System Design and Implementation

In this section, we present the design and implementation of the PerfScope system. We first describe the offline function signature extraction scheme. We then describe the online bug inference algorithms.

5.2.1 Offline Function Signature Extraction

We combine dynamic binary instrumentation with frequent episode learning techniques to extract the function signatures without requiring application source code. Our scheme constructs robust function signatures by learning what closed frequent system call episodes\(^3\) can be generated by each function. Unlike other alternative approaches using low level system metrics (e.g., execution time, CPU usage) that often vary significantly among different platforms, we found closed frequent system call episodes remain stable under different workloads and resource allocation settings as shown by our experiments. Thus, we can perform function signature offline outside the production environment without requiring the exactly same workload or environment that trigger the production-run performance bug.

Binary-based Function Signature Profiling

To learn what system calls are produced by each function, we correlate the function execution list provided by the dynamic binary instrumentation tool and the system call trace provided by LTTng based on the timestamp information\(^4\).

The function execution list records all the function entry points and exit points from an

---

\(^3\)Closed frequent episodes mean that we only consider the maximum frequent episodes, not any sub-sequence of those frequent closed episodes.

\(^4\)Although we can also use the dynamic binary instrumentation tool (e.g., the PIN tool) to collect the system call trace directly, those tools do not provide the function origin of each system call by themselves.
executing application. Each entry in a function execution list is in the form \{time stamp, enter function \(f_i\), thread ID\} or \{time stamp, exit function \(f_i\), thread ID\}. For C/C++ programs, we use the Pin tool \cite{76} to insert the logging code into the binary to record when a function is called and when it returns with a timestamp. For Java programs, we insert hooks into the Java Virtual Machine (JVM) to intercept function invocations as they are being executed, logging when each function is called and when it returns with a timestamp.

Each system call entry consists of a pair of logs: 1) \[timestamp\text{entry}\] system call name: \{procname, ppid, process ID, thread ID, cpu ID\} and 2) \[timestamp\text{exit}\] sys \_exit: \{procname, ppid, process ID, thread ID, cpu ID\}. We can derive the system call execution time by computing the difference between timestamp \(T_{\text{entry}}\) and timestamp \(T_{\text{exit}}\). Figure 5.2 shows an excerpt of real system call trace from a Tomcat application server run.

![Function execution list and System call trace](image)

Figure 5.3: Function signature profiling. By combining the function execution list and the system call trace via the active function stack, we label each system call with the function that produces it.

We then create a dynamic stack of active functions for each running thread to determine which function was active when each system call was generated. When a function is called, we push the function onto the corresponding active function stack based on the thread ID associated with the function. When a function returns, we pop the function off the active function stack. Given a system call \(s_i\) with a thread ID \(tid = k\), we compare its exit timestamp \(T_{s_i}\) with the timestamp \(T_{\text{top}}\) of the top function \(f_i\) on the active function stack for the same thread \(tid = k\). We also compare \(T_{s_i}\) with the timestamp \(T_{\text{next}}\) of the next function \(f_{i+1}\) in the function execution list. If the system call \(s_i\) occurs after the application enters \(f_i\) but before enters \(f_{i+1}\) (i.e., \(T_{s_i} \geq T_{\text{top}}\) and \(T_{s_i} < T_{\text{next}}\)), we know \(s_i\) must have been generated by \(f_i\). If we find the exit timestamp of the current system call \(s_i\) is larger than or equal to the timestamp of the next function in the function execution list (i.e., \(T_{s_i} \geq T_{\text{next}}\), we push the next function \(f_{i+1}\) onto the stack and label that the system call \(s_i\) is produced by \(f_{i+1}\). This process continues
Figure 5.4: Frequent closed system call episodes discovery. All the frequent system call episodes (counter >10) are highlighted in gray color. The frequent closed episode is highlighted within a circle.

until either we have no more functions to process or no more system calls in the system call trace. An example of this process is shown in Figure 5.3. As shown, since $T_{top} \leq T_{s1} < T_{next}$, we infer $s_1$ is produced by the function $f_2$ that is currently on the top of the active function stack.

After the annotations for all the system calls are done, we group all the system calls produced by the same function together and sort them based on their timestamps into time ordered sequences. For example, if the system call `sys_write` annotated with function $f_i$ is followed by `sys_read` annotated with the same function, we add the sequence `{sys_write, sys_read}` into the function profiling result for $f_i$. If a function consists of any branch statements, the profile of the function may consist of a set of system call sequences (e.g., {{sys_write, sys_read}, {sys_poll, sys_futex}}) when different branch paths are executed.

**Frequent System Call Episode as Robust Function Signature**

Although we could use the set of the raw system call sequences to represent the signature of the function, we found that those raw system call sequences are sensitive to execution environment changes. Since our goal is to find a robust function signature that is stable under different execution environments, we extract *closed frequent system call episodes* from those raw system call sequences, which can better characterize the key distinct behaviors of different applications. During our experiments, we observe that those frequent system call episodes are quite stable under different execution environments.

We extract closed frequent system call episodes from the raw system call sequences for each function using a common frequent episode mining algorithm called the *A-priori* method [16,91].
The basic idea of the A-priori method is to start at the basic event type (e.g., a single system call) and then build up more complex sequences by taking all possible permutations using all the frequent episodes at all lower levels (i.e., frequent episodes of smaller size), shown by Figure 5.4.

The frequent episodes search iterates until a maximum level is reached where no frequent episode can be found. We perform pruning to reduce the number of items to be processed at each level. The intuition behind our pruning scheme is that a frequent episode at a higher level must be a frequent episode at a lower level. So it is only necessary to generate the permutations of the episodes we marked as frequent in the previous level. A minimum support value ($L$) is usually used to define the frequency lower bound for generating frequent episodes.

Figure 5.4 illustrates the frequent closed system call sequence discovery algorithm using a minimum support value $L = 10$. We start from three single system call episodes. We then drop `sys_poll` since its count does not meet the minimum support. At the level 2, we search all the permutations of the frequent episodes in the level 1 and identify `(sys_write, sys_read)` as a frequent episode. Moving up the level 3, we consider the permutations of all the frequent episodes found in the level 1 and 2. We found four possible permutations and none of them meets the minimum support. We stop the frequent episode search and outputs the `(sys_write, sys_read)` as the frequent closed episode.

We implement frequent closed episode frequency counting using finite state machine (FSM) based matching algorithm. During the candidate generation phase, we build a FSM for each candidate episode. Each state in the FSM represents a system call type. Given an input system call sequence, we match this sequence with all the candidates’ FSMs simultaneously. When the input system call matches the current state in the FSM, we move the match pointer to the next state in the FSM. If the input system call does not match, we reset the match pointer to the start state. When all the states in the FSM are matched, we increase the frequency counter of the candidate episode represented by this FSM by 1 and reset the match pointer to the start state of the FSM.

In our experiments, we found that if we only consider contiguous frequent episodes, we often only discover trivial system call sequences that cannot help us distinguish different functions. Thus, we modified our algorithm to discover non-contiguous frequent episodes such as `(sys_write, *, sys_read)`, where * represents arbitrary system calls or zero system call. To discover non-contiguous frequent episodes, we only need to make a minor change to the above matching algorithm: if the input system call does not match the current state in the FSM, we do not reset the match pointer to the start state and wait until the next match occurs.

How to set a proper minimum support value ($L$) is a tricky issue. We calculate the minimum support value using the standard way of multiplying the number of items (i.e., system calls included in the execution unit $N_{sys\_call}$) with a certain percentage $p$ [133], that is $L = N_{sys\_call} \times \ldots$
In our experiments, we use \( p = 1\% \). We also conduct the sensitivity study on other possible values of \( p \) to evaluate its impact to our system. The general principle behind the minimum support value selection is that we should not use an overly high minimum support value, which prevents us from finding any frequent episodes; we should also avoid using a trivial minimum support value (e.g., \( L = 1 \)) as this causes us to find too many frequent episodes (e.g., tens of thousands). So we set \( L = \max(\min(N_{\text{sys-call}} \times p, 10), 2) \) to cap the minimum support value at 10 and avoid the trivial minimum support value. We found our calculation scheme works well for all different systems we tested.

It can be time-consuming to perform the signature extraction for a complex server program consisting of tens of thousands of functions. We observe that the function signature extraction for different functions are independent from each other and can run in parallel. So we implement a distributed function signature extraction system to speed up the profiling process. Moreover, the function signature extraction is done offline, which will not affect our online bug inference time.

Note that PerfScope also does not require to profile all different function call paths but just individual functions. The function call path of the buggy run is inferred from the system call trace of the production-run directly, which will be explained in Section 5.2.2. We can also selectively profile a subset of user defined functions and exclude those uninteresting functions such as system libraries. For example, for C/C++ programs, we can use indexing tools such as cTags [4] to generate the list of profiled functions. For Java programs, we can use package names (e.g., Hadoop, catalina) to select the functions to be profiled.

### 5.2.2 Online Bug Inference

Our online bug inference scheme consists of three major steps: 1) extracting different execution units from the raw system call trace; 2) identifying anomalous execution units; 3) mapping anomalous execution units into a rank list of bug related functions. We will describe each step in detail in this section.

#### Execution Unit Extraction

The system call trace collected by LTTng is a massive stream of mixed system calls produced by different processes, threads, and functions. To identify bug-related functions, we first need to segment the raw system call trace into a set of execution units. Ideally, each execution unit should consist of a set of semantically related system calls which are produced by the same function, thread, and process. Since the system call trace collected by LTTng provides process ID and thread ID, it is easy to segment the trace into different threads/processes. However, during experiments, we found that the thread-based segmentation is sometimes insufficient.
First, we observe that some server systems (e.g., Apache web server) use a pre-allocated thread pool in order to avoid the overhead of constantly creating new threads. The threads in the thread pool are reused for different tasks, which can cause the behavior of those threads to vary significantly over time. Second, some long running threads often execute multiple different functions. To address this issue, we propose to use the time gap to split those threads. The intuition is that a large time gap between two consecutive system calls in a thread implies a thread recycling or function switching. Therefore, we compute the intervals between each pair of contiguous system calls in the trace and use the mean plus 2 times standard deviation as the segmentation threshold to split the per-thread trace. We found this segmentation threshold works well for all the tested systems and also conducted sensitivity study on this threshold value in the experiments.

During dynamic application runtime, we need to address one issue with the time gap based segmentation. We observe that the context switch between different threads caused by CPU scheduler sometimes creates a time gap between two semantically related system calls. Thus, we need to ignore those time gaps caused by the context switch. Luckily, it is easy to detect context switches in the LTANT collected system call trace. For example, when we detect either a thread ID change or a CPU ID change, we know a context switch has happened.

**Abnormal Execution Unit (AEU) Identification**

We use light-weight unsupervised learning methods to identify abnormal execution units. We chose unsupervised learning approach because it does not require any labelled training data which are hard to obtain the production environment. Specifically, we use a hybrid anomaly detection algorithm that combines hierarchical clustering and outlier detection.

We first apply a top-down hierarchical clustering [63] to group those execution units that perform similar operations together. This grouping is important, which can not only increase our accuracy but also reduce the processing time as the outlier detection step is more costly. For this purpose, we extract a system call appearance vector for each execution unit. The length of the system call appearance vector is equal to the number of unique system call types (e.g., `sys_write`, `sys_poll`) seen across all the execution units. Each system call type is assigned a position in the vector. The value is set to 1 if that type of system call is present in the execution unit, or 0 if it is not. For example, consider two execution units, `{sys_write, sys_read}` and `{sys_poll, sys_read}`. In this case, system call appearance vector follows the format of `[sys_write, sys_read, sys_poll]`. The system call appearance vector for the two execution units would be `[1, 1, 0]` and `[0, 1, 1]`, respectively. We chose the system call appearance vector as the clustering feature vector to achieve robust grouping since two similar execution units might produce different numbers of the same system call type due to dynamic runtime
environments.

Next, we perform outlier detection within each cluster to identify abnormal execution units. We observe that the abnormal behaviors of the affected execution units may manifest in both system call frequencies or execution time. For example, a loop bug may cause certain system calls to be executed more frequently whereas a synchronization bug may cause the lock acquiring system calls to take longer to complete their execution. Therefore we build a system call execution time vector (consisting of average execution time for each system call type) and a system call frequency vector (consisting of the count for each system call type) as two features for each execution unit. We use the nearest neighbor algorithm \[108\] to perform the outlier detection. Specifically, we compare the Euclidean distance from each execution unit to its nearest neighbor within the cluster using either the execution time vector or the frequency vector. If the nearest neighbor distance of an execution unit is larger than the mean nearest neighbor distance of the whole cluster plus two times the standard deviation, we say this execution unit is abnormal. We observe that some execution units might form a small cluster (e.g., size of the cluster < 4) by themselves. It is not meaningful to perform outlier detection within those small clusters. All the execution units in those small clusters are considered to be abnormal.

**Bug Related Function List Generation**

We now describe how we map those abnormal execution units to bug-related application functions and rank them based on an abnormality degree metric.

For each abnormal execution unit \(AEU_i\), we extract a set of frequent closed system call episodes using the same frequent episode mining algorithm described in Section 5.2.1. Let \(FE_{AEU_i}\) denote the set of the frequent closed system call episode set produced by \(AEU_i\). Similarly, our offline function signature profiling process produces a set of frequent closed system call episode \(FE_{f_i}\) for each function \(f_i\). If any frequent episode in \(FE_{AEU_i}\) matches with any frequent episode in \(FE_{f_i}\), we infer that \(AEU_i\) must execute part of the function \(f_i\) and output \(f_i\) as one candidate function.

For each candidate function \(f_i\), we compute a total count of all the matching frequent episodes for \(f_i\) and its corresponding abnormal execution unit \(AEU_i\), respectively. If the total frequent episode count of \(AEU_i\) is greater than or equal to that of \(f_i\), we infer that \(AEU_i\) must have run some parts of \(f_i\) incorrectly (e.g., an incorrect loop) when the bug is triggered. We then output \(f_i\) as one identified function.

We then calculate a rank score for each identified function using a maximum percentage increase metric (i.e., the largest count increase percentage among all the matched frequent episodes between \(AEU_i\) and \(f_i\)) to quantify the abnormality degree of different functions during the buggy run. We sort all the identified functions using increasing rank scores. We break the
<table>
<thead>
<tr>
<th>System name</th>
<th>Num of bugs studied</th>
<th>% of bugs generating system calls</th>
<th>% of slowdown bugs</th>
<th>% of hang bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>40</td>
<td>83%</td>
<td>10%</td>
<td>90%</td>
</tr>
<tr>
<td>HDFS</td>
<td>32</td>
<td>100%</td>
<td>38%</td>
<td>62%</td>
</tr>
<tr>
<td>Cassandra</td>
<td>47</td>
<td>85%</td>
<td>28%</td>
<td>72%</td>
</tr>
<tr>
<td>Tomcat</td>
<td>22</td>
<td>95%</td>
<td>18%</td>
<td>82%</td>
</tr>
<tr>
<td>Apache</td>
<td>17</td>
<td>100%</td>
<td>41%</td>
<td>59%</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>10</td>
<td>90%</td>
<td>30%</td>
<td>70%</td>
</tr>
<tr>
<td>MySQL</td>
<td>60</td>
<td>90%</td>
<td>38%</td>
<td>62%</td>
</tr>
</tbody>
</table>

5.3 Experimental Evaluation

We tested PerfScope using seven popular open source server systems: 1) Hadoop [2]: a map-reduce framework; 2) HDFS [2]: a distributed file system; 3) Cassandra [19]: a distributed relational database system; 4) Tomcat [20]: a multi-threaded application server; 5) Apache web server [18]: a multi-threaded web server; 6) Lighttpd [71]: a light-weight web server; and 7) MySQL [79]: a relational database system. In this section, we first describe our static bug analysis result and the bug samples we could reproduce and use in our experiments. We then present our experiment setup followed by the bug inference results. We then discuss our sensitivity study results followed by the overhead evaluation.

---

Some of the results presented in this chapter differ from the published version of PerfScope due to a more thorough understanding of the bugs obtained through detailed analysis using HSR. We present HSR in the following chapter.
Table 5.2: Reproduced real C/C++ performance bugs that are used in our experiments.

<table>
<thead>
<tr>
<th>Bug name</th>
<th>Server description (LOC)</th>
<th>Num. of error msg</th>
<th>Num. of threads</th>
<th>Num. of system calls (million)</th>
<th>Log size (MB)</th>
<th>Bug description</th>
<th>Bug symptom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-1</td>
<td>Web server (161K C/C++)</td>
<td>0</td>
<td>19</td>
<td>3.7</td>
<td>142</td>
<td>Incorrect flag causes infinite loop. (#37680)</td>
<td>hang</td>
</tr>
<tr>
<td>Apache-2</td>
<td>Web server (194K C/C++)</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>140</td>
<td>Incorrect flag causes proxy SSL connections to be constantly created and destroyed, slowing down performance. (#43238)</td>
<td>slowdown</td>
</tr>
<tr>
<td>Lighttpd-1</td>
<td>Web server (38K C/C++)</td>
<td>0</td>
<td>10</td>
<td>5.3</td>
<td>160</td>
<td>Incorrect errno handling causes an infinite loop. (#1212)</td>
<td>hang</td>
</tr>
<tr>
<td>Lighttpd-2</td>
<td>Web server (37K C/C++)</td>
<td>0</td>
<td>18</td>
<td>4.3</td>
<td>532</td>
<td>Large chunks of response data are repeatedly read and discarded while processing header information. (#1999)</td>
<td>slowdown</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>Database server (914K C/C++)</td>
<td>0</td>
<td>551</td>
<td>1.9</td>
<td>138</td>
<td>Two threads try to execute the INSERT DELAYED statement but one of them has a locked table, causing two threads to become deadlocked. (#54332)</td>
<td>hang</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>Database server (1.3M C/C++)</td>
<td>0</td>
<td>9</td>
<td>1.9</td>
<td>39</td>
<td>Flushing to disk abnormally frequently after truncating a large table. (#65615)</td>
<td>slowdown</td>
</tr>
</tbody>
</table>
Table 5.3: Reproduced real Java performance bugs that are used in our experiments.

<table>
<thead>
<tr>
<th>Bug name</th>
<th>Server description (LOC)</th>
<th>Num. of error msg</th>
<th>Num. of threads</th>
<th>Num. of system calls (million)</th>
<th>Log size (MB)</th>
<th>Bug description</th>
<th>Bug symptom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>Data processing framework (1.3M Java)</td>
<td>0</td>
<td>729</td>
<td>1.5</td>
<td>222</td>
<td>Endless wait for an atomic variable to be set. (#MAPREDUCE-3738)</td>
<td>hang</td>
</tr>
<tr>
<td>HDFS</td>
<td>Data processing framework (1.3M Java)</td>
<td>1</td>
<td>340</td>
<td>1.4</td>
<td>227</td>
<td>Continuously read until timeout on a socket. (#HDFS-3318)</td>
<td>hang</td>
</tr>
<tr>
<td>Cassandra</td>
<td>Database (168K Java)</td>
<td>0</td>
<td>95</td>
<td>1.1</td>
<td>241</td>
<td>Incorrect return value handling causes Cassandra to enter an infinite loop. (#5064)</td>
<td>hang</td>
</tr>
<tr>
<td>Tomcat-1</td>
<td>Application server (287K Java)</td>
<td>0</td>
<td>20</td>
<td>0.2</td>
<td>18</td>
<td>Tomcat tries to upgrade a read lock to a write lock, which causes the server to hang. (#53450)</td>
<td>hang</td>
</tr>
<tr>
<td>Tomcat-2</td>
<td>Application server (284K Java)</td>
<td>1</td>
<td>682</td>
<td>0.9</td>
<td>68</td>
<td>A shared counter value is not updated correctly causing all the threads to keep checking the value endlessly. (#53173)</td>
<td>hang</td>
</tr>
<tr>
<td>Tomcat-3</td>
<td>Application server (228K Java)</td>
<td>0</td>
<td>26</td>
<td>0.4</td>
<td>28</td>
<td>A filter chain item is not set properly causing an infinite loop. (#42753)</td>
<td>hang</td>
</tr>
</tbody>
</table>
Table 5.4: Online bug inference result summary.

<table>
<thead>
<tr>
<th>Bug name number of functions, percent of identified functions</th>
<th>Top ranked bug-related functions (Rank)</th>
<th>Examples of detected abnormal application-kernel interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop 85495, 0.52%</td>
<td>getState (4)</td>
<td>Continuously produces {sys gettimeofday, sys stat64, sys stat64, sys gettimeofday} for acquiring locks to check an atomic variable.</td>
</tr>
<tr>
<td>HDFS 81306, 1.5%</td>
<td>Reader.performIO (3), SocketIOWithTimeout.doIO (4)</td>
<td>Continuously produces {sys gettimeofday, sys read, sys read, sys gettimeofday} for attempting to perform I/O.</td>
</tr>
<tr>
<td>Cassandra 21765, 0.6%</td>
<td>maybeSwitchMemtable (4), SSTableSliceIterator.hasNext (5)</td>
<td>Continuously produces {sys gettimeofday, sys futex, sys futex, sys gettimeofday} for synchronization from a buggy infinite loop.</td>
</tr>
<tr>
<td>Tomcat-1 32789, 1.5%</td>
<td>addLifecycleListener (2)</td>
<td>Continuously produces {sys stat64, sys gettimeofday, sys gettimeofday, sys stat64} for upgrading from a read lock to a write lock.</td>
</tr>
<tr>
<td>Tomcat-2 32571, 0.8%</td>
<td>LimitLatch.countDown (11)</td>
<td>Continuously produces {sys gettimeofday, sys stat64, sys stat64, sys gettimeofday} while checking an atomic counter value.</td>
</tr>
<tr>
<td>Tomcat-3 26471, 0.4%</td>
<td>run (3)</td>
<td>Continuously produces {sys gettimeofday, sys read, sys read, sys gettimeofday} while reading and attempting to process an event in a buggy infinite loop.</td>
</tr>
<tr>
<td>Apache-1 17721, 0.7%</td>
<td>proc_mutex.sysv.acquire (8)</td>
<td>Continuously produces {sys gettimeofday, sys write} for logging in a buggy infinite loop.</td>
</tr>
<tr>
<td>Apache-2 18761, 2%</td>
<td>ssl_hook_pre_connection (9)</td>
<td>More {sys stat64, sys write, sys gettimeofday, sys stat64} are generated for logging in the buggy ssl_hook_pre_connection function.</td>
</tr>
<tr>
<td>Lighttpd-1 954, 0.1%</td>
<td>fdevent_poll (1)</td>
<td>Continuously produces {sys poll, sys time, sys time, sys poll} while waiting for events to be processed in a buggy infinite loop.</td>
</tr>
<tr>
<td>Lighttpd-2 3577, 0.08%</td>
<td>connection_handle_read_state (3)</td>
<td>Continuously produces {sys ioctl, sys read, sys read, sys ioctl} while repeatedly reading large chunks.</td>
</tr>
<tr>
<td>MySQL-1 25360, 0.6%</td>
<td>fil_aio_wait (3)</td>
<td>Continuously produces {sys stat64, sys write, sys write, sys stat64} when trying to write over the network.</td>
</tr>
<tr>
<td>MySQL-2 34850, 0.03%</td>
<td>buf_flush_list (1)</td>
<td>More {sys futex, sys gettimeofday, sys gettimeofday, sys futex} are generated for synchronization while flushing lists to disk.</td>
</tr>
</tbody>
</table>
5.3.1 Real Performance Bug Samples

We searched the bug repositories of each system we tested using performance terms (e.g., hangs, slowdown, etc.) and developer provided performance tags when available. We only examined those bugs which were confirmed by developers and subsequently fixed. Table 5.1 shows the results of our study. We manually examined each bug, along with any provided patch, to determine what kind of problem the bug caused (e.g., hang, slowdown) and whether the buggy function generated system calls. We found that over 90% bugs generate system calls. We found more hang bugs than slowdown bugs in each system.

We then try to reproduce each bug by following the instructions in the report and checking that the expected performance anomaly symptoms appeared (e.g., increased response time, 100% CPU usage, system is unresponsive). 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. Table 5.2 and 5.3 list the 12 performance bugs we could reproduce given our time limit for C/C++ and Java systems, respectively.

Although we only reproduced 12 out of the 228 bugs due to time constraints, the characteristics of the bugs we reproduced are similar to the other bugs we studied. Specifically, we found most of the bugs we examined involve some kind of loop problem, which is consistent with the observations of previous work [58, 84]. However, the bug points do not always reside in those loops. As we will show later, PerfScope can identify those bug-related functions that either include the bug points or are close to the functions that include the bug points. We found that several system hangs were the result of an application component waiting for a response which will never arrive or executing a blocking command without an appropriate timeout value. These types of problems can be difficult to track down. We also found the fix to these types of hang bugs was typically simple (e.g., adding timeout value). Lastly, we found I/O (e.g., file write) and locking operations were the most common system call generating operations. Based on our study, we believe the majority of the system call generating bugs we examined can be correctly handled by PerfScope.

5.3.2 Experiment Setup

The Hadoop, HDFS, Tomcat, and Cassandra systems were tested on a cloud test bed where each host is equipped with a quad-core Xeon 2.53Ghz CPU along with 8GB memory and runs 64bit CentOS 5.3 with KVM 0.12.1.2. The Apache, Lighttpd, and MySQL systems were tested on the virtual computing lab (VCL) [15], a production cloud infrastructure where each host has a dual-core Xeon 3.0GHz CPU and 4GB memory, and runs 64bit CentOS 5.2 with Xen 3.0.3. In both cases, each system call trace was collected in a virtual machine using LTTng 2.0.1 running 32-bit Ubuntu 12.04 kernel version 3.2.0.
We use PIN version 2.12 [76] to generate the function signature profiles for Apache, MySQL, and Lighttpd written in C/C++. For Hadoop, HDFS, Tomcat, and Cassandra, we instrumented the openJDK6 JVM [88] to perform the function signature profiling. The function signature extraction were done under the following workload conditions: 1) Hadoop: we use the Pi calculation application with 16 map and 16 reduce tasks; 2) HDFS: we transferred a 200MB file using hftp; 3) Cassandra: we use a simple workload which creates a table and inserts various entries into the table; 4) Tomcat: we randomly request different example servlets and JSPs included with Tomcat following a workload intensity observed in a NASA web server trace [14]; 5) Apache: we use httperf to request various pages from the Apache server; 6) Lighttpd: we use the same workload as Apache and configure Lighttpd to run in multi-process mode; 7) MySQL: we use an open source MySQL benchmark tool called Sysbench [13] and did the oltp test in complex mode. Note that the workloads we used for profiling are different from the bug-triggering workloads. We also intentionally vary the resource allocation setting for the profiling environment to test the robustness of our function signatures. We found that the workload and resource allocation changes did not significantly affect the frequent episodes generated. For example, varying the workload intensity from 50 static page requests per second to 125 static page requests per second while building an Apache profile only caused 4% of the generated frequent episodes to change.

During each buggy run, PerfScope is triggered when the performance anomaly is detected by an external detector that checks for reported performance anomaly symptoms such as zero progress score or abnormal response time. Once triggered, PerfScope was run using the recent 1.5 minutes of system call traces for analysis. We can retrieve longer system call trace from an NFS server if needed. However, our experiments show that 1.5 minutes system call trace is sufficient for all the bugs we tested.

5.3.3 Bug Inference Result Summary

Table 5.4 shows a summary of our bug inference results for all tested bugs. We can see most server systems we tested consist of tens of thousands of functions. As shown in Tables 5.2 and 5.3, most performance bugs do not produce any error message and some buggy runs also involve a large number of threads, it is a daunting task for the developer to figure out why the performance bugs occurred. Profiling by itself can help reduce the search scope as we only profile the functions called the system configuration described in the bug report, which was 0.7% to 10% of the total functions. However, this still equates to thousands of functions to search. The results show that PerfScope can identify a short list of bug-related functions for all tested bugs, which can greatly reduce the search scope to only 0.03-2.3% of total functions. We believe that this search scope reduction can significant expedite the debugging process.
To further validate the effectiveness of our bug inference results, we manually examined the source code of each tested server system to determine whether each of our short lists indeed includes the bug-related functions which can help the developer to locate the bug. We list the top ranked bug-related functions identified by PerfScope in Table 5.4. We found that the short list of identified functions indeed cover the bug-related functions in all the bugs we tested. We rank the bug-related function within the top five candidates in 9 out of the 12 bugs. In the worst case, we rank the bug-related function at 11. When multiple functions contribute to the problem (e.g., HDFS bug, Cassandra bug), PerfScope can identify them as well. Table 5.4 also shows one example of abnormal frequent system call episodes detected by PerfScope for each bug.

As shown in our results, PerfScope can handle both performance degradation bugs and hang bugs. PerfScope works best for detecting those bugs involving a buggy loop (e.g., an infinite loop or a loop iterates more times than usual). Previous work [58] along with our own study have shown most performance bugs do exhibit these characteristics.

We also investigated the cases where PerfScope fails to rank the bug-related function within top 5 and identified two main reasons causing this. First, we found there are some frequent system call episodes which are common to several functions. When the abnormal frequent system call episodes also appear in a function called less frequently than the bug-related function during profiling, this can cause us to rank that function higher than the bug-related function. However, we found most of those mistakenly high-ranking functions (e.g., initialization functions) can be easily filtered out by the developer. Second, we sometimes give high rank to common utility functions (e.g., custom fast comparison function) used throughout the application. When those utility functions are called several times in the buggy function, this can cause the utility function to have a higher rank score than the buggy function. We are currently exploring ways of filtering out those common utility functions.

5.3.4 Bug Inference Case Studies

To further understand how the output of our bug inference tool can be used for debugging, we now discuss three of our bug inference results in detail. Due to the space limitation, we cannot describe the inference results for all the bugs in detail. We hope that these three case studies can show the usefulness of PerfScope for helping the developer diagnose the production-run performance bugs.

**Cassandra bug:** This Cassandra performance bug is related to a memtable mishandling problem. Figure 5.5 shows a subgraph of the bug’s call graph. The root cause of this bug is the if statement at line 650 of the maybeSwitchMemtable function. The code block (lines 651 and 652) within the if statement should be executed at least once for every memtable object.
in order to add the memtable object into the memtable list. However, if a memtable is clean, the check at line 650 will be false, causing the memtable not to be added to the list. As the memtable has not been processed, the check at line 172 in the reload function never becomes true, causing the program never breaks out of the while loop. No error message is produced for this bug.

PerfScope analysis produced 1229 execution units, which were clustered into 49 clusters. Our outlier detection results identified 19 time based outlier units and 33 frequency based outlier units. PerfScope successfully identifies the root cause function maybeSwitchMemtable function as the fourth top ranked function. We found this function produced a set of 30 frequent episodes during profiling. We also found that this function is only called in a limited number of locations. Thus, identifying this function can help the developer quickly localize the problem. The fix to this bug is to force the code block within the if statement at line 650 of the maybeSwitchMemtable function to be executed at least once.

**HDFS bug:** The second bug we discuss in detail is a HDFS bug mentioned in the introduction, which is caused by an overflow of an internal variable. Figure 5.1 shows the subgraph of the bug’s call graph. When copying a file using the distcp command, the copyBytes function is called to read 4096 bytes from the source stream and write those bytes to the destination stream.
This causes the read, doIO, and performIO functions to be called in turn. The performIO function then makes a low level socket call and returns the number of bytes read back. This process is repeated until -1 is returned, signaling the end of the input stream has been reached. When this command is called with a file larger than 2GB in size, however, an internal variable (int) representing the length of the source stream overflows and the end of stream signal (i.e., -1) is never sent. This causes while loop at line 76 of the copyBytes function to never end until the operation times out several minutes later. The only error message generated throughout this process is a message saying the operation timed out. Additionally, as HDFS uses many different interfaces and abstract classes, it is difficult to identify which implementation subclass causes the problem.

PerfScope analysis produced 3817 execution units, which were clustered into 174 clusters. Our outlier detection results identified 92 time based outlier units and 113 frequency based outlier units. The third top ranked function PerfScope identified is the SocketInputStream$Reader.performIO function. We found this function produced a set of 30 frequent episodes during profiling. As shown in Figure 5.1, this is one of the functions directly responsible for reading data from the input stream and is directly related to the bug. By telling developers not only the function name, but also telling the developer that the bug is related to the SocketInputStream$Reader class, debugging time for this problem can be greatly reduced. Additionally, identifying that the problem is caused by an infinite loop could help developers immediately rule out other misleading causes, such as a network issue. The fix in this case is to ensure streams larger than 2GB is processed in chunks of appropriate size.

**Lighttpd-1 bug:** The third bug we want to discuss is a Lighttpd bug shown by Figure 5.6. The performance anomaly is caused by the if statement at line 1282 of the main function always stays true, causing the code block within the if statement executes forever. During this buggy run, the fdevents function is first invoked at the if statement at line 1282 of the main function. Next, the fcgi_handle_fdevent function is invoked at line 1309, which then calls the fcgi_demux_response function. The root cause of this problem is an improper errno handling (i.e., setting errno to EAGAIN) at line 2442 of the fcgi_demux_response function causes this function to return 0. When the function fcgi_handle_fdevent receives a 0 return value from fcgi_demux_response, it does not clean up this event from the event queue. This in turn causes fdevent_poll to return a value > 0, which causes the system to process this same event again and again. No error message is generated during this process.

PerfScope analysis produced 9380 execution units, which were clustered into 7 clusters. Our outlier detection results identified 33 time based outlier units and 31 frequency based outlier units. PerfScope identifies the fdevent_poll as the top ranked bug-related function, which controls the event processing code at line 1282 of the main function. We found this function produced a set of 900 frequent episodes during profiling. The large number of frequent episodes
Figure 5.6: A subset of the call graph for the Lighttpd bug. The errno is handled incorrectly, causing the main function to process the same event again and again. The bug point is highlighted in bold. PerfScope identified the `fdevent_poll` as the bug related function which is responsible for controlling the event processing to continue as long as the number of events (`n`) is positive.

produced by this function are a result of the variety of tasks it is involved with. By starting debugging at this function, developers could quickly see that the same event was being processed over and over. This information is especially useful for Lighttpd, which has a complex main function responsible for performing many different tasks. Furthermore, the function we identify resides in the same `if` statement code block with the root cause function. This means developers would have significantly fewer areas to explore when trying to debug the problem. The fix for this issue would be to ensure `errno` is handled appropriately, ensuring the event is processed.

5.3.5 Sensitivity Study

We conducted sensitivity study experiments to evaluate how different parameter settings affect our bug inference results. Although we omit the results due to space limitations, we found that the parameter values do not significantly affect our bug inference results. We also examined the ranks of the true bug related functions, which also show little changes.

5.3.6 PerfScope Overhead

We now evaluate the overhead of PerfScope to different server systems. For Hadoop/HDFS we ran the Pi sample application. For Cassandra, we ran a database insertion workload. We used
httpperf to send a number of requests to Tomcat, Apache, and Lighttpd. We used a request rate of 50 requests per second for Tomcat, 100 requests per second for Apache and Lighttpd. For MySQL, we run a constant workload of 20 select requests per second. Those request rates are set based on the maximum processing capacity of our host. We ran all overhead experiments 5 times, reporting the mean. We impart an average of 1.8% runtime overhead to the server system. Specifically, we impart 2.97% to Hadoop/HDFS, 3.33% to Cassandra, 1.4% to Tomcat, 0.8% to Apache, 0.3% to Lighttpd, and 2.2% to MySQL. We found PerfScope imparts between 2-3% CPU load and has a small memory footprint (about 256KB). We also found the storage overhead is reasonable, varying between about 18MB to 530MB.

We also ran an experiment to determine how PerfScope overhead scales under different workload intensity by comparing the overhead of tracing at 50, 75, 100, and 125 requests per second to an Apache web server. We found the log size scales linearly with request rate while the overhead of tracing remained constant. We used least squares linear regression to project log sizes for larger request rates. We found that the one-minute tracing log size will be around 450MB given 900 requests per second. Note that this log size can significantly reduced using standard compression techniques. The low overhead we observed is consistent with the overhead results of previous projects [75]. Hence, we believe that PerfScope is light-weight and practical for online bug inference in production cloud computing infrastructures.

We now present the online and offline computation time of PerfScope shown by Table 5.5. The results show that PerfScope can complete the online bug inference within tens of minutes. The majority of the online inference time was spent on extracting the frequent episodes for

Table 5.5: Online and offline computation time for PerfScope.

<table>
<thead>
<tr>
<th>Bug name</th>
<th>Online bug inference</th>
<th>Offline function signature extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>15.99 min</td>
<td>9.5 min</td>
</tr>
<tr>
<td>HDFS</td>
<td>29.94 min</td>
<td>38 min</td>
</tr>
<tr>
<td>Cassandra</td>
<td>3.34 min</td>
<td>12.9 min</td>
</tr>
<tr>
<td>Tomcat-1</td>
<td>5.14 min</td>
<td>42.7 min</td>
</tr>
<tr>
<td>Tomcat-2</td>
<td>12.34 min</td>
<td>2.9 hour</td>
</tr>
<tr>
<td>Tomcat-3</td>
<td>2.32 min</td>
<td>48.7 min</td>
</tr>
<tr>
<td>Apache-1</td>
<td>2.93 min</td>
<td>15.5 min</td>
</tr>
<tr>
<td>Apache-2</td>
<td>4.3 min</td>
<td>14.4 min</td>
</tr>
<tr>
<td>Lighttpd-1</td>
<td>9.91 min</td>
<td>1.02 hour</td>
</tr>
<tr>
<td>Lighttpd-2</td>
<td>5.87 min</td>
<td>1.15 hour</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>12.8 min</td>
<td>44.5 min</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>5.22 min</td>
<td>44.5 min</td>
</tr>
</tbody>
</table>
different anomalous execution units in order to identify the bug-related functions. This time can be further shortened if we use multiple distributed hosts to perform the frequent episode mining. Our distributed offline function signature extraction time ranged from 9.5 minutes to about 2.9 hours as shown in Table 5.5. These times were obtained by running our distributed signature extraction algorithm on a 7 node cluster and could be further improved on larger clusters. Additionally, as the function signature profiling is performed offline outside the production environment, its overhead will not be a concern for using PerfScope to perform online bug inference.

5.4 Summary

In this chapter, we have presented PerfScope, a novel online performance debugging tool for production hosting infrastructures. PerfScope achieves faulty function localization without requiring source code or runtime instrumentation. Instead, PerfScope uses kernel-level system call tracing to capture the abnormal behaviors of the performance bug. To bridge the gap from the kernel space to the application space, PerfScope uses offline profiling to extract the signature of each function in the form of frequent closed system call episodes. During runtime, PerfScope can quickly map the abnormal execution behaviors to different faulty functions using the offline extracted function signatures. We have implemented the PerfScope system and conducted experiments using a set of popular open source systems and a range of different real world performance bugs. The results show that PerfScope can successfully localize the root cause related functions by ranking them in the top 0.001-2.3% of all functions. PerfScope is light-weight, which only imposes 1% average runtime execution overhead to the monitored applications.
Chapter 6

HSR: Hybrid Static-Runtime Analysis for Performance Anomaly Diagnosis

In this chapter we discuss the final component in our framework, HSR. PerfScope is able to automatically produce a ranked list of functions affected by a performance anomaly at runtime. However, many of the functions identified are false positives. HSR is designed to combine static analysis with this runtime information in order to provide more accurate diagnosis information. We begin by introducing the work and then discuss HSR in detail.

6.1 Introduction

Diagnosing performance anomalies caused by internal software bugs is a challenging task. Static rule-based checking work [58] has been done to identify problems at compile time. However, these approaches suffer from false positives because it is difficult to differentiate functions with buggy characteristics (i.e., code segments that may cause bugs) from actual buggy functions. Dynamic diagnosis tools [42] which can identify functions affected by a performance anomaly at runtime also exist. These approaches also suffer from many false positives due to misidentified functions unrelated to the bug.

In the context of this work, we refer to any non-bug related functions presented as output to end-users as false positives. To illustrate why false positives occur, we discuss Apache-37680. This bug was discovered when a user made one simple change to the configuration file. First, the user started the web-server with a configuration to listen on two ports. Next, the user changed the configuration file to only listen to one port. Finally, the user restarted the server using the graceful restart option. The graceful restart option attempts to minimize any downtime by
only restarting the parts of the application requiring a restart. Instead of the web server coming back online as normal in a few seconds as expected, the user found that the server was hung, consuming 100% CPU in the process.

The root cause of this bug was a blocking call Apache attempts to make on a reused socket. A graceful restart reuses the sockets from the previous running instance without clearing any flags set on the sockets. In this bug, the 0_NONBLOCK flag was previously set on the socket, preventing it from making blocking calls. When Apache tries to make the blocking call without clearing this flag, it fails. Although the socket call function returns an integer value indicating whether the call succeeded, Apache simply checks if the call result was APR_SUCCESS, retrying otherwise. Because the flag will never be cleared, this causes the web-server to loop, continuously re-trying the call.

```c
if (ap_listeners && ap_listeners->next) {
    for (lr = ap_listeners; lr; lr = lr->next) {
        apr_status_t status;
        status = apr_socket_opt_set(lr->sd, APR_SO_NONBLOCK, 1);
        if (status != APR_SUCCESS) {
            ap_log_perror(APLOG_MARK, APLOG_STARTUP | APLOG_ERR, status, pool,
                           "ap_listen_open: unable to make socket non-blocking");
            return -1;
        }
    }
    use_nonblock = (ap_listeners && ap_listeners->next);
    for (lr = ap_listeners; lr; lr = lr->next) {
        apr_status_t status;
        status = apr_socket_opt_set(lr->sd, APR_SO_NONBLOCK, use_nonblock);
        if (status != APR_SUCCESS) {
            ap_log_perror(APLOG_MARK, APLOG_STARTUP | APLOG_ERR, status, pool,
                           "ap_listen_open: unable to control socket non-blocking status");
            return -1;
        }
    }
}
#endif /* AP_NONBLOCK_WHEN_MULTI_LISTEN */
```

Figure 6.1: The patch for Apache-37680. The bug occurs as a result of the constant value 1 being passed to the apr_socket_opt_set function, causing an endless loop in another function at runtime.

The patch to fix the bug is given in Figure 6.1. As the figure shows, there is only one major change to the application, other than the logging message change. Instead of a constant value
being passed to the `apr_socket_opt_set` function, we see a variable named `use_nonblock` is passed to the function. This variable controls whether a blocking or non-blocking call should be made. On a graceful server restart, this variable is set to allow non-blocking calls to be made on the socket and preventing the infinite loop.

A simple static rule for this bug would be to check if a constant value is passed to a function as a parameter. In fact, we found this pattern actually exists in several of the bugs we studied. The problem with doing this, however, is most of the time when a constant value is passed to a function, there is no bug. Thus, functions with buggy characteristics are not necessarily buggy and are instead, false positives. Similarly, system call based runtime diagnosis tools [42] may identify logging functions such as `ap_log_perror` as bug related functions. However, more often than not those types of functions are not related to any bug but are simply on the call path of buggy loops, making them false positives.

In this chapter we present HSR, a hybrid static-runtime analysis tool for performance anomaly diagnosis. HSR works using two phases. First, we build an application profile offline which uses static analysis to automatically identify functions with buggy characteristics. Online, we then combine this profile with runtime diagnosis information to provide a short list of buggy functions for developers to investigate.

Specifically, we make the following contributions:

- We show how generic rules can be developed to identify functions with buggy characteristics. These functions typically are more complex than their non-buggy counterparts.

- We show how runtime information can be combined with static analysis in order to provide more accurate diagnosis information compared to purely runtime and purely static diagnosis approaches.

We have implemented HSR and have tested it using 9 real-world software bugs spanning five open source systems (HDFS, Tomcat, Apache, Lighttpd, MySQL). Our results indicate that HSR can significantly reduce the number of matched functions for developers to analyze by up to 98% compared to purely static approaches and by up to 91% compared to purely runtime approaches while also covering bug-related functions. Furthermore, by analyzing bitcode and bytecode, HSR does not require any source code to perform static analysis.

The rest of the chapter is organized as follows. Section 6.2 describes the design of the HSR system. Section 6.3 presents our experimental evaluation. Finally, the chapter concludes in Section 6.4.
6.2 Design

In this section we discuss the design details of HSR. We begin with an overview of our system. We then discuss our rule-based static analysis module and our hybrid function identification module.

6.2.1 System Overview

Figure 6.2 shows an overview of HSR. As shown there are two main components of our system, a rule-based static analysis module and a hybrid function identification module. The rule-based static analysis module is responsible for performing rule based static analysis over the target application offline to create a possibly buggy function profile. Our hybrid function identification module then uses the output of dynamic analysis tools (e.g., PerfScope) along with this profile to identify functions with buggy characteristics which are also experiencing a runtime problem.
6.2.2 Rule-Based Static Analysis

Developing rules for static analysis is a challenging task. On one hand, very specific rules can be written to identify known bugs, but those rules are often not able to identify other similar bugs. On the other hand, more generic rules can be developed to identify potential bugs, but those rules are often too generic to give additional useful diagnosis information to developers.

We choose to take the latter approach to developing rules. That is, we aim to develop more generic rules to identify functions with buggy characteristics. We choose to take this approach because it allows us to develop a relatively small amount of rules that can be automatically applied to many systems instead of having to manually develop specific bug rules for each individual system. This also allows HSR to work with previously unknown bugs as opposed to looking for previously identified bugs. Furthermore, we believe that experienced developers already have a deep understanding of the systems they are developing and providing a short list of potentially buggy functions can significantly reduce diagnosis time.

Pattern Identification

In order to develop rules for HSR to use, we have to first identify a statically checkable pattern to look for. Statically checkable patterns differ from dynamically checkable patterns in the fact that they must be checkable without runtime information. For example, checking whether a function call contains a constant value as a parameter is statically checkable. Checking whether a pointer to a variable has taken on a particular value at runtime is an example of a dynamically checkable rule.

To find these patterns, we manually inspect patches in existing bug reports contained in the bug reporting databases for each system. Because prior work [41, 53], has identified hang bugs as significant challenges for cloud systems, we have designed our rules based around identifying those types of bugs. We found that many bugs have very specific fixes, making identification of generic, statically checkable patterns a challenging task. Previous work [58] found similar results, building 25 static checkers for the 109 patches overall they examined. In addition, the checkers they developed looked for specific problems and were not designed to identify generic statically checkable patterns existing across multiple systems. Fortunately, we were able to identify several statically checkable patterns which exist across multiple systems.

The “Constant value passed to a function” rule we have developed is good example to show how the patterns for these rules are identified. For this rule, we found the pattern that constant (i.e., hard coded) values as parameters can cause unexpected results. For example, Figure 6.3 shows the patch for MySQL bug 28000. In this bug, the constant hard-coded value of 0 causes MySQL to never ignore errors when executing the fill_record_invoke_before_triggers. In most cases, this is not a problem as errors should be handled appropriately (e.g., logged).
However, in certain circumstances, such as when executing the `INSERT IGNORE` command, errors should be ignored. As a result of errors not being ignored under the conditions given in the bug report, the system ultimately hangs.

For this pattern, we actually found bugs with similar patterns in four (Apache, MySQL, HDFS, Tomcat) out of the five systems we tested. However, as these are generic buggy characteristics, we may or may not see reported patches in all systems. It depends if users have triggered a condition for the bug to manifest at runtime. As a result, we do not require explicitly that a pattern show up in multiple bug reports and instead only require that the identified pattern be generalizable to other systems. For example, we do not consider patterns which involve functions that exist on a single system (e.g., `child_main`). The pattern we found for our “Constant value passed to a function” rule is a good pattern to use because both function invocations and constants exist across all systems. Thus, checking whether a function invocation has been called with a constant value is generalizable across different systems.

```c
if (fill_record_n_invoke_before_triggers(thd, + *info->update_fields, + *info->update_values, 0, + *info->update_values, info->ignore, table->triggers, TRG_EVENT_UPDATE))
```

Figure 6.3: The patch for MySQL bug 28000. The bug occurs as a result of the constant value 0 being passed to the `fill_record_n_invoke_before_triggers` function, causing an endless loop at runtime.

**Rule Generation**

Once we identify a candidate buggy pattern, we use static analysis of bitcode or bytecode to produce the detection rules. We choose to operate at this level as opposed to source-level because it allows HSR to work in situations where source code may not be available due to issues such as privacy concerns. Operating at this level, however, has some issues associated with it. For example, in Java, function calls are converted to the `invokedynamic` instruction, with the arguments being the previous $n$ instructions before it. In this case, $n$ is the number of arguments for that particular function. An example of this is illustrated in Figures 6.4 and 6.5. Specifically, Figure 6.5 is the result of decompiling source code shown in Figure 6.4. In this example, our detection rule would be to look for a constant instruction (e.g., `iconst_1`) in the argument instructions appearing before the `invokevirtual` instruction.
public class Test{
    public void FunctionA(){
        FunctionB(1);
    }
    public int FunctionB(int param){
        return param+1;
    }
}

Figure 6.4: Simple Java source code showing FunctionA calling FunctionB.

public void FunctionA();
Code:
0: aload_0
1: iconst_1
2: invokevirtual #2 // Method FunctionB:(I)I
5: pop
6: return

public int FunctionB(int);
Code:
0: iload_1
1: iconst_1
2: iadd
3: ireturn

Figure 6.5: The result of decompiling the simple source code shown in Figure 6.4. Function call to FunctionB is transformed into an invokedynamic instruction with the parameters being the iconst_1 instruction before it.

In order to address the challenges of operating at the bytecode or bytecode level, we have developed several custom extensible bug detectors using existing static analysis frameworks. In addition to providing an interface to existing checker functionality, our checkers provide several utility functions, such as invokedynamic argument extraction. Additionally, we have designed HSR to work using a simple generic message output template shown in Figure 6.6. This allows users to quickly and easily add new rules to HSR as required with few code changes. Our possibly buggy function profile is a collection of these messages for a particular application.

6.2.3 Hybrid Function Identification

The insight behind HSR is that we can identify three types of functions, as shown by Figure 6.7. First, we can identify functions with buggy characteristics, but no runtime issues. This is the
Figure 6.6: The message template for HSR. Items shown in brackets allow users to quickly develop new rules for HSR with few code changes.

Figure 6.7: The type of functions we can identify. The functions likely to cause a bug are in the intersection of the two sets shown. That is, those functions with buggy characteristics that have experienced a problem at runtime.

left portion of the “Static Analysis Matched Functions” set shown in Figure 6.7. This could be due to the fact that they simply are not being used at runtime (e.g., dynamically loaded modules). These are false positives generated by purely static analysis. Second, we can identify functions executing at runtime without any buggy characteristics. This is the right portion of the “Runtime Identified Functions” set shown in Figure 6.7. This could be because they are simply non-buggy functions on the execution path of a buggy function. These are false positives generated by purely runtime analysis. Lastly, we can identify functions with buggy characteristics which are also experiencing a problem at runtime. This is the intersection of the “Static Analysis Matched Functions” set and the “Runtime Identified Functions” set shown in Figure 6.7. The intuition behind our approach is that functions with buggy characteristics which are also experiencing a problem at runtime are more likely than not to be related to the root cause of a bug.

Our hybrid function identification module is designed to take an existing possibly buggy function profile and combine that with functions identified at runtime. Specifically, we only consider functions in the middle section of Figure 6.7. That is, we only output a function if it appears in both the runtime output list and the possibly buggy function profile. When we find a function which exists in both, we output the function name along with its rank and the buggy characteristics the function has.

In addition to running HSR to output runtime affected functions with buggy characteristics,
we have developed two modes which allow HSR to output additional diagnosis information. These two modes are based on the intuition that sometimes the caller of a function can be related to a performance bug. Many trivial bugs involving a single function are caught with simple unit testing. However, more complex call chains involving multiple functions interacting are more difficult to test. When debugging, knowing the context information in which a function was called can help to debug these complex interactions. In addition, some higher level functions do not generate many system calls. By including the immediate context of the functions generating system calls, it is possible to identify those higher level functions which may be contributing to the problem as well. For example, in the Apache bug case discussed in Section 6.1, we do not identify the child_main function in our runtime log because it does not generate many system calls. Thus, although we output other bug related functions, we do not output the child_main function. We do however, identify the proc_mutex_sysv.acquire function, which is called by child_main. Therefore, by including caller information it is possible to identify this function.

As shown by Figure 6.8, most functions in an application can be called in different possible ways. For example, Function F can be called be either Function C, Function D, or Function E. Using static analysis, we can identify all the potential call paths for a given function. Some of these call paths, however, may or may not be present at runtime due to the environment (e.g., user input, application configuration). Only at runtime is it possible to know the actual call path taken by an application. In this example, the call path taken at runtime is Function A → Function B → Function C → Function F. We now discuss our two context modes in detail.
Figure 6.9: The function invocation message template for HSR. This template allows HSR to be easily extended to perform a variety of control flow analysis as needed.

**Static Call Graph Mode**

The first mode we have developed to provide additional diagnosis information is static call graph mode. In this mode, while performing static analysis to identify buggy characteristics we also record all the possible ways each function can be called. Specifically, while analyzing the definition of each function in an application, when we see an instruction corresponding to a function invocation (e.g., invokedynamic), we output a message to record the caller and callee. To output our results, we use a generic message template similar to the message template used for buggy functions. Our message template for function invocations is shown in Figure 6.9. In this case, we output the `call` option to indicate to HSR that this is a statically analyzed possible function invocation.

When static call graph mode is enabled, we first identify all the runtime functions which also have buggy characteristics as usual. For each identified function, however, we also include all the caller functions of each matched runtime function in our results. We rank each caller function, in order processed, as the rank of the identified runtime function + 1. For example, if we rank Function F from Figure 6.8 as rank 1, we would rank Function C, Function D, and Function E as ranks 2, 3, and 4, respectively. Although this gives more detailed diagnostic information and generates less false positives compared to static analysis only, we found that this mode can generate many false positives.

**Runtime Call Path Mode**

The second mode we have developed is runtime call path mode. In this mode, we identify the different ways each function is called at runtime. To do this, we take a function trace as input for the application we are interested in monitoring and simulate the stack for each thread. This function trace is the same function trace used by PerfScope, described in Chapter 6. As different functions are pushed on to the stack, we record which function (i.e., the stack top) called that function. Figure 6.10 shows an example of how this is done. As shown, when function at time $T_{current}$ is pushed on to the stack, we output a message indicating that the function at $T_{top}$ called function at $T_{current}$.

When runtime call path mode is enabled, we output the caller functions of each function
Figure 6.10: The call stack for a traced application for thread tid = 5. When a function at time \( T_{current} \) is pushed on to the stack we output a message indicating that the function at \( T_{top} \) called function at \( T_{current} \). We then increment \( T_{current} \) to the next entry.

along with each of the callers of that function we have identified at runtime. To output our results, we use the same call message template used to output static caller information. However, in this case, we output the \( \texttt{r-call} \) option to indicate this is a runtime call. We rank the functions in a similar way as with static context mode ranking. We found this mode is capable of producing more detailed diagnostic information at the cost of some additional false positives.

6.3 Evaluation

In this section we discuss the evaluation of our prototype implementation of HSR. We begin by discussing the details of our implementation. We then discuss our results.

6.3.1 Implementation

We have implemented a prototype of our system using Python and have tested it with several bugs collected in our previous studies. To perform static analysis, we use LLVM [10] for C/C++ applications and Findbugs [5] for Java applications. Our implementation consists of about 750 lines total of C/C++, Java, and Python. We now discuss our implementation and results in detail.

C/C++

To perform static analysis on C/C++ applications, we have developed code transformations using LLVM. LLVM is a compiler infrastructure which supports a wide variety of languages such as C, C++, and Objective-C. In addition, LLVM allows developers to examine or modify code as it is being compiled by providing a programmatic interface to the intermediate representation.
used by LLVM for an application. These transformations are called passes and can modify the code in a variety of ways. For example, the FunctionPass class allows developers to inspect and modify the code for each function in a compiling application.

We have implemented our analysis as LLVM passes. Specifically, we have implemented FunctionPass and LoopPass transformations. The FunctionPass transformation identifies issues existing in function definitions. The LoopPass transformation identifies issues existing with loop definitions.

**Java**

For Java applications, we implemented our transformations using Findbugs. Findbugs is an application which is designed to analyze Java bytecode to find instances of bug patterns using the Apache BCEL [3]. Findbugs comes with a variety of built in patterns to identify common bugs (e.g., bad coding practice). In addition, Findbugs also allows users to write custom bug detectors in the form of plugins. We have implemented our Java static analysis as a custom Findbugs plugin. This plugin can be used directly on a target directory or can be easily integrated into the build process of the target application.

**Rules**

We have implemented five rules based on our analysis of the various bugs we studied shown in Table 6.1. Each of these rules were developed by manually examining the patches to identify the buggy segment of code causing a performance problem. We found that developing static rules for all bugs was not possible.

In addition to identifying the functions in an application with characteristics that have been known to cause bugs in the past we have also found in some cases that the rules themselves are useful in providing additional diagnosis clues to developers. We discuss this further in Section 6.3.4.

**6.3.2 Experiment Methodology**

We evaluated HSR using real performance bugs affecting various cloud server systems. These bugs were selected by searching the bug repositories of each system using performance terms (e.g., hangs, slowdown, etc.) and developer provided performance tags when available. When a candidate bug was identified, we followed the instructions in the report to reproduce the bug and checked that the expected performance anomaly symptoms appeared (e.g., increased response time, 100% CPU usage, system is unresponsive). As previous work [42] has also noted, bug reproduction is extremely time-consuming and tricky due to limited and often ambiguous information. It can sometimes take up to a month to reproduce one bug.
Table 6.1: Descriptions for the static rules we tested.

<table>
<thead>
<tr>
<th>Rule</th>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsafe function: fopen</td>
<td>r1</td>
<td>Call to fopen can cause a problem if file is larger than 2GB on 32-bit machines.</td>
</tr>
<tr>
<td>Unsafe function: atol</td>
<td>r2</td>
<td>Call to atol can cause a problem if file is larger than 2GB on 32-bit machines.</td>
</tr>
<tr>
<td>Unconditional loop:</td>
<td>r3</td>
<td>Unconditional loop in function (e.g., while(1)).</td>
</tr>
<tr>
<td>Constant Arg</td>
<td>r4</td>
<td>Constant hard coded argument passed to function. Can cause bug under unexpected configurations.</td>
</tr>
<tr>
<td>Null Arg</td>
<td>r5</td>
<td>Null argument passed to function. If incorrectly handled, can cause performance bug.</td>
</tr>
</tbody>
</table>

Table 6.2: Descriptions of the real-world bugs tested.

<table>
<thead>
<tr>
<th>Bug name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>Continuously read until timeout on a socket.</td>
</tr>
<tr>
<td>Tomcat-2</td>
<td>A shared counter value is not updated correctly causing all the threads to keep checking the value endlessly.</td>
</tr>
<tr>
<td>Tomcat-3</td>
<td>A filter chain item is not set properly causing an infinite loop.</td>
</tr>
<tr>
<td>Apache-1</td>
<td>Incorrect flag causes infinite loop.</td>
</tr>
<tr>
<td>Apache-2</td>
<td>Incorrect flag causes proxy SSL connections to be constantly created and destroyed, slowing down performance.</td>
</tr>
<tr>
<td>Apache-3</td>
<td>Calling fopen on file larger than 2GB causes hang on 32-bit systems.</td>
</tr>
<tr>
<td>Lighttpd-1</td>
<td>Incorrect errno handling causes an infinite loop.</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>Two threads try to execute the INSERT DELAYED statement but one of them has a locked table, causing two threads to become deadlocked.</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>Flushing to disk abnormally frequently after truncating a large table.</td>
</tr>
</tbody>
</table>
Table 6.2 describes the 9 real bugs we reproduced and tested. When we successfully reproduced a bug, we collected the runtime information. We then ran HSR on the bytecode or bitcode of the application to collect a possibly buggy function profile for that application. Finally we feed both the runtime information and profile into our hybrid function identification module to produce a HSR buggy function list.

To examine the effectiveness of HSR at reducing false positives, we used three different approaches on each bug. First, we performed static analysis alone using the five rules we defined on each bug to identify how many functions our generic rules would match. Next, we reproduced each bug and ran PerfScope, described in Chapter 6, on them in order to identify how many runtime functions we can identify. Finally, we ran HSR on each bug, combining the static analysis and runtime results.

We evaluated our system in four major aspects. First, how frequently do the rules we identified generate false positives? Second, can we improve the ranking results for the bug-related functions compared to a purely runtime approach? Third, what reduction in false positives, if any, can we produce by combining runtime feedback with static analysis compared to purely static and runtime-based approaches? Finally, are the rules matched in each function useful for helping developers diagnose the problem?

6.3.3 Rules Matched

We now discuss the number of times each rule was output by HSR across all the bugs we tested along with the number of false positives caused by each rule. Our results are shown in Table 6.3. These results are what we expected to see. Specifically, we saw that the more generic rules, such as rules involving constants, were matched the most frequently. Rules which were more specific, such as those rules matching specific functions (e.g., `fopen`) were matched less frequently.

The most commonly matched rule was rule r4, a constant hard coded argument passed to a function. This is expected as constant values are commonly passed to functions for control flow. We also found that rule r2, `atol` causing a problem for files larger than 2GB, was unmatched for the bugs we tested. Although this rule is unmatched for the bugs we tested, it is possible it may match different versions of the software. As a result we keep the rule in our set.

The number of false positives caused by each rule follows the same trend. As shown, the generic rules, such as rules r4 and r5, generate more false positives. This is because most of the time, when passing NULL or a constant value to a function, there is no problem. The more specific rules generate fewer false positives. This is because those rules are developed by looking at specific bugs. Therefore, when they are matched, they are likely to be the cause of a bug.
Table 6.3: The total number of times each rule is matched across all bugs we tested.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Total Matches</th>
<th>Total False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>r2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>r3</td>
<td>47</td>
<td>45</td>
</tr>
<tr>
<td>r4</td>
<td>1683</td>
<td>1674</td>
</tr>
<tr>
<td>r5</td>
<td>1092</td>
<td>1087</td>
</tr>
</tbody>
</table>

Table 6.4: Results of combining runtime and static analysis for diagnosis.

<table>
<thead>
<tr>
<th>Bug name</th>
<th>Best PerfScope bug related function rank</th>
<th>Best HSR bug related function rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Tomcat-2</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>Tomcat-3</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Apache-1</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Apache-2</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Apache-3</td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td>Lighttpd-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

6.3.4 Ranking Results

Table 6.4 shows the ranking of the best bug-related function identified by PerfScope and HSR, respectively. As shown, we see that the HSR output-rank of the bug-related function increases significantly compared to the PerfScope rank for almost all the bug cases we tested. This indicates that many of the runtime matched functions are likely simple helper functions which are unlikely to be related to the bug. The one case we do not see an increase is in the Tomcat-2 case. Upon close examination, we found this is because the runtime-identified bug related function `run`, does not match any static rules. Overall, these results indicate that the runtime ranking results can be improved by combining runtime and static analysis.

Table 6.5 shows the functions we match using the LLVM/Findbugs only, PerfScope only, and HSR schemes, respectively. In most cases we found that we match the same functions using all three schemes, indicating the number of false positives is the major trade off to using the different schemes. We show the effectiveness of each scheme at lowering the amount of false positives generated in Section 6.3.5.

In addition to these results we also evaluated HSR using the static call graph and runtime call path modes. We found these two modes are able to provide additional useful diagnosis.
Table 6.5: The bug related functions we identified using LLVM/Findbugs, PerfScope analysis, and HSR schemes.

<table>
<thead>
<tr>
<th>Bug name</th>
<th>LLVM/Findbugs bug-related functions</th>
<th>PerfScope bug-related function</th>
<th>HSR functions</th>
<th>Matched rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>doIO, copyBytes, read</td>
<td>performIO, doIO, read</td>
<td>doIO, copyBytes, read</td>
<td>r3,r4</td>
</tr>
<tr>
<td>Tomcat-2</td>
<td>countDown</td>
<td>countDown</td>
<td></td>
<td>r4</td>
</tr>
<tr>
<td>Tomcat-3</td>
<td>processSocket</td>
<td>run, processSocket</td>
<td>processSocket</td>
<td>r4,r5</td>
</tr>
<tr>
<td>Apache-1</td>
<td>child_main, proc_mutex, sysv_acquire, apr_socket_opt_set</td>
<td>proc_mutex, sysv_acquire</td>
<td>proc_mutex, sysv_acquire</td>
<td>r4,r5</td>
</tr>
<tr>
<td>Apache-2</td>
<td>ssl_init_ssl_connection, ssl_hook_pre_connection</td>
<td>ssl_hook_pre_connection</td>
<td>ssl_hook_pre_connection</td>
<td>r4,r5</td>
</tr>
<tr>
<td>Apache-3</td>
<td>err_output, log_err</td>
<td>err_output, log_err</td>
<td></td>
<td>r1,r4</td>
</tr>
<tr>
<td>Lighttpd-1</td>
<td>fd_event_poll, fcgi_handle_fdevent, main</td>
<td>fd_event_poll</td>
<td>fd_event_poll</td>
<td>r3,r4,r5</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>fil_aio_wait, fil_aio_wait, ha_lock_engine</td>
<td>fil_aio_wait</td>
<td>fil_aio_wait</td>
<td>r4,r5</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>mi_read_static_record, mi_read_static_record, buf_flush_list</td>
<td>mi_read_static_record</td>
<td>mi_read_static_record</td>
<td>r4</td>
</tr>
</tbody>
</table>

information for the Apache-1 and Lighttpd cases. In particular, we found the functions identified with these two modes enabled were able to identify the same functions we identified using static analysis only, but with fewer false positives. We now discuss each case in detail.

**HDFS:** In this case static analysis identifies different functions compared to the PerfScope results. Specifically, we found the performIO function did not have any static issues. The static rule we have identified for the best bug-related function are rules r3 and r4. Rule r3 is useful for developers because this bug involves endlessly reading from a socket in a loop. Rule r4 is not particularly useful in providing additional feedback to developers in this case.

**Tomcat-2:** The same function is identified by the LLVM/Findbugs, PerfScope, and HSR approaches. The rule identified is not particularly useful in providing additional runtime feedback to developers. The rule does, however, correctly identify functions that could contribute to the problem.

**Tomcat-3:** In this case, we identify more functions with PerfScope compared to LLVM/Findbugs only. This is because we found the Poller.run function generated system calls and was thus matched at runtime. However, that function did not have any static issues. The rules are not helpful in providing additional diagnosis feedback.

**Apache-1:** For this bug, LLVM/Findbugs analysis identified an additional bug-related function, child_main, which was not identified in our HSR results. This is because the function was not present in the list of functions identified at runtime, because the child_main function does not
produce many system calls. However, when run under both static call graph mode and runtime call path mode, HSR is able to also identify the `child main` function. The matched rules are not helpful in providing additional diagnosis feedback.

**Apache-2:** LLVM/Findbugs analysis identified an additional bug-related function which was not identified by our HSR results. As before, this is because the function was not identified at runtime. The rules matched in this case are not useful for diagnosis. This is because the bug involves initializing new SSL connections as opposed to reusing the same connection when applicable as the result of a flag being set incorrectly during server initialization. The type of rule we could develop for this bug is not generalizable. The matched rules in this case correctly identify non-trivial functions capable of causing performance problems.

**Apache-3:** The same functions were identified by the LLVM/Findbugs, PerfScope, and HSR approaches. Rule `r1` provides additional useful diagnosis information to developers. Specifically, this rule identifies the function that is causing the bug. Rule `r4`, however, does not provide additional useful diagnosis information for developers.

**Lighttpd-1:** LLVM/Findbugs analysis identified additional bug-related function, `main`, which was not identified by HSR. As before, this is because those functions do not produce many system calls. The static rules are not very helpful for providing additional diagnosis information. In this case, we also see a very large reduction in the number of matched functions. This is because Lighttpd is specifically designed to have relatively few functions and we only identify one buggy function at runtime. When run using static call graph mode, HSR is able to also identify the bug related function `main`. When run in runtime call path mode, HSR is unable to identify any additional bug related functions. The rule identified for the `main` function, rule `r3`, is helpful for diagnosis as it correctly indicates there is an unconditional loop.

**MySQL-1:** In this case, we identify more functions with PerfScope compared to LLVM/Findbugs analysis only. This is because one of the runtime identified functions, `ha lock engine` does not match any static rules. It does, however, generates system calls which is why it is identified in our runtime list. The matched rules are not helpful in providing additional diagnosis feedback.

**MySQL-2:** Similar to the MySQL-1 case, we identify more functions at runtime with PerfScope compared to LLVM/Findbugs analysis only because the `buf flush list` does not match any of our static rules. As before, it does generate system calls which is why it appears in the runtime list. The matched rules are not helpful in providing additional diagnosis feedback.

### 6.3.5 Reduction Results

Figures 6.11 and 6.12 show the results of running our framework on the real Java and C/C++ bugs we tested, respectively. We also show the percent reduction HSR achieves in false positives for each bug in Table 6.6. As shown, combining runtime and static analysis is able to significantly
reduce the number of false positives developers examine. Specifically, we are able to reduce the number of false positive functions for developers to analyze by up to 98% compared to LLVM/Findbugs analysis only and 91% compared to PerfScope only while also identifying bug-related functions (i.e., true positives). These results indicate that many of the functions identified by static analysis and at runtime are indeed false positives unrelated to the bug affecting the application.

Figures 6.11 and 6.12 also show the reduction in false positives with static call graph and runtime call path modes enabled for the Java and C/C++ bugs, respectively. These context modes were useful in some of our results for identifying additional bug-related functions but as shown, this additional diagnosis information comes at the cost of additional false positives.

6.4 Summary

In this chapter we have discussed HSR, a hybrid static-runtime analysis tool for performance bug diagnosis in cloud systems. Our results show that HSR is capable of reducing the number
Figure 6.12: The number of false positives identified for each scheme for each C/C++-based bug we tested. As shown, combining runtime analysis with static analysis causes significantly fewer false positives to be identified while also identifying bug-related functions.

of functions for developers to analyze by up to 98% compared to purely static approaches and by up to 91% compared to purely runtime approaches while also covering bug-related functions. Lastly, HSR does not rely on source code for static analysis.
Table 6.6: The percent reduction in false positives achieved when combining runtime and static analysis for diagnosis.

<table>
<thead>
<tr>
<th>Bug name</th>
<th>Percent reduction vs. LLVM/Findbugs</th>
<th>Percent reduction vs. PerfScope</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>98%</td>
<td>91%</td>
</tr>
<tr>
<td>Tomcat-2</td>
<td>91%</td>
<td>79%</td>
</tr>
<tr>
<td>Tomcat-3</td>
<td>94%</td>
<td>79%</td>
</tr>
<tr>
<td>Apache-1</td>
<td>96%</td>
<td>42%</td>
</tr>
<tr>
<td>Apache-2</td>
<td>76%</td>
<td>50%</td>
</tr>
<tr>
<td>Apache-3</td>
<td>96%</td>
<td>48%</td>
</tr>
<tr>
<td>Lighttpd-1</td>
<td>99%</td>
<td>0%</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>97%</td>
<td>53%</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>99%</td>
<td>82%</td>
</tr>
</tbody>
</table>
Chapter 7

Related Work

In this chapter we describe the works related to our work. We begin by discussing previous work done to predict performance anomalies. We then discuss work done to localize performance anomalies. Next we discuss work done to diagnose performance anomalies. Finally, we discuss works using static, dynamic, and hybrid static-dynamic analysis to identify bugs.

7.1 Performance Anomaly Prediction

Work has been done using supervised approaches for anomaly prediction. The most closely related work to ours is Tiresias [124], which also addresses the black-box failure prediction problem in distributed systems. Tiresias relies on external anomaly detectors to create anomaly vectors. The system then applies Dispersion Frame Technique (DFT) prediction heuristics on the anomaly vectors for anomaly prediction. Gu et al. [52] integrate Markov feature value prediction with naive Bayesian classification to predict performance anomalies. Tan et al. [110] use a hierarchical clustering technique to discover different execution contexts of a dynamic system but build context-aware prediction models to improve prediction accuracy. Different from UBL, the above works need labelled normal and failure data in the training data and do not provide anomaly cause inference. In contrast, UBL does not require any data labeling, which allows UBL to predict both known and unknown performance anomalies.

Supervised techniques have been previously used for anomaly detection. Cohen et al. [39] use clustering over labelled failure data to extract failure signatures, which can be used to detect recurrent problems. Powers et al. [93] study different statistical learning methods to find the approaches that can detect performance violations in an enterprise system. Bhatia et al. [28] develop sketches of system events, which are then visualized for diagnosis by an expert. The Fa system [46] uses anomaly-based clustering to achieve automatic failure diagnosis for query processing systems. Cha et al. [32] use a signature based approach along with a
bloom filter for malware detection. Bodik et al. [29] use signatures along with feature selection and a regression model to detect performance anomalies. In contrast, our approach focuses on predicting unknown anomalies and does not require prior knowledge about different failure instances.

Unsupervised approaches have been previously used to perform anomaly detection. Previous work has proposed model-driven approach to performance anomaly detection. For example, Stewart et al. [104] instrument the OS to gather data and profile system performance using queuing models. Shen et al. [99] use a reference based approach to detect performance anomalies by looking at how metrics differ from the ideal case. Stewart et al. [103] use a transaction mix model to predict the performance given a certain workload, and hence can detect the anomaly if the observed performance is different with the predicted performance. Compared to UBL, those model-driven approaches typically require extensive model calibration using offline profiling and need to make certain assumptions about the workload type (e.g., transactions) and user request arrival patterns. In contrast, UBL is application-agnostic and does not require extensive application profiling. Cherkasova et al. [35] build regression-based transaction models and application performance signatures to provide a solution for anomaly detection considering system changes. Different from UBL, this model is designed to consider a single metric. Wang et al. [117] have used entropy based approaches to quantify the metric distribution and detect anomalies using signal processing and spike detection. Similarly, Jiang et al. [57] detect failures by looking at the entropy of clustered system metric relationships. Makanju et al. [77] assign entropy scores to event log data to detect anomalies. WAP5 [96] detects the bottleneck in distributed systems by analyzing message traces to infer the causal structure and timing of communication within these systems. Kasick et al. [62] use peer comparison to determine the root cause of a problem in a distributed environment. Jiang et al. [56] employ linear regression models to extract invariants and then track their changes to detect the anomaly in transaction systems. In contrast to the above approaches, UBL can predict future anomalies as opposed to detecting anomalies at the moment of failure. Moreover, UBL is broader in scope, designed to learn system behavior for a variety of uses. We show anomaly prediction as one of the uses of UBL.

7.2 Performance Anomaly Localization

Previous work has proposed to develop performance debugging tools using system event traces such as system system call traces and hardware performance counters. Fournier et al. [49] collected system level event traces and built a state transition model to analyze the blocking behavior on multi-core systems. Ding et al. [44] used system calls and command line options to build normal application signatures for diagnosing previously known problems. Shen et
al. [99] construct change profiles to detect system anomaly symptoms by checking performance deviation between reference and target executions. DeepDive [85] first clusters low-level metrics to identify potential interference in virtualized environments. Potential interferences are then confirmed or denied by comparing the VM performance in isolation with that in production. Abhishek et al. [98] uses Auto-Regressive model to detect time-invariant relationships from monitoring data, and analyzes broken invariants during runtime to localize the fault. Lan et al. [68] use principle component analysis and independent component analysis along with outlier detection to automatically find faulty components in large-scale systems. In contrast, our work focuses on distinguishing faults with a globally direct impact from faults with a locally direct impact, which is critical for efficiently diagnosing performance anomalies in shared hosting infrastructures. Moreover, our approach does not assume any prior knowledge about the diagnosed problem. Thus, we can diagnose emergent production problems that are previously unseen.

Fay [47] uses probes, which are kernel drivers or DLLs, along with hotpatching to collect and summarize detailed user-mode and kernel level traces without modifying the underlying system. Although Fay is able to effectively determine performance problems, it does so through a combination of user space and kernel level tracing, imparting up to 280% overhead in the process. In contrast, our approach relies on only kernel level events, imparting 0.8-3.3% overhead in doing so. Thus, our approach is more suitable for diagnosing production-run problems within the large-scale hosting infrastructure.

Magpie [26] instruments middleware and packet transfer points in order to generate a series of low level network events. They then group these events together into requests via a series of temporal joins and finally cluster together requests based on behavior. DARC [111] identifies functions that are the main latency contributors to peaks in a OSProf profile. $S^2E$ [37] uses selective symbolic execution and dynamic binary instrumentation to perform in-vivo multi-path analysis for finding bugs and profiling performance. The above approaches require instrumentations to the application or the middleware, which is difficult to be deployed in the production hosting infrastructure. In contrast, PerfCompass is designed to run in the production hosting infrastructure without any application or middleware instrumentation.

Researchers also proposed white-box approach to analyzing performance bugs. Jin et al. [58] present a study of 109 real world performance bugs in 5 different systems. By manually checking the patches of known problems, they are able to then build an efficiency rule-based checker which was able to identify previously unknown performance problems in deployed software. Cloud-Diag [78] automatically identifies and ranks anomalous function calls using robust principle component analysis along with white-box instrumentation to provide fine-grained performance debugging information to developers. However, it is difficult to obtain the source code access for production systems running inside the cloud hosting infrastructures. Thus, PerfCompass
strives to perform online performance anomaly localization and inference without requiring source code.

Previous work has analyzed RPC messages to identify faulty components. For example, Aguilera et al. [17] analyze RPC traces to infer causal paths for debugging. Similarly, WAP5 [96] links logical messages between different components in distributed systems in order to identify causal paths for debugging. In contrast, PerfCompass uses system call traces to achieve fine-grained fault localization within each faulty component identifying whether the fault comes from the environment or the component software itself and extracting hints about the fault. Previous black-box fault localization schemes cannot provide that type of fine-grained diagnosis.

The idea of tracing distributed systems and networks for debugging has been well studied [25, 33, 34, 48, 54, 95, 97, 128]. These systems perform network level tracing in order to infer paths or dependency information for diagnosing large-scale distributed systems. In comparison, our work focuses on fine-grained root cause inference within a single server, which is complementary to those network tracing approaches.

Much work has been done to diagnose configuration problems. X-ray [22] uses Pin to dynamically instrument binaries to track the information flow in order to estimate the cost and the likelihood that a block was executed due to each potential root cause (e.g., configuration option). Autobash [105] proposed to use the pattern of success and failure of known predicates to diagnose misconfigurations. Chronus [122] periodically checkpoints disk state and automatically searches for configuration changes that may have caused the misconfigurations. PeerPresure [118] and Strider [120] identify differences in configuration states across several different machines to diagnose misconfigurations on an affected machine. Our work is complementary to the above work by focusing on performance anomalies caused by environment issues or internal software bugs.

Record and replay techniques [51, 67, 102] are useful for system performance diagnosis. TRANSPLAY [106] uses partial checkpointing and deterministic record-replaying to capture minimum amount of data necessary to reexecute just the last few moments of the application before it encountered a failure. Triage [112] leverages checkpoint-replay to perform just-in-time problem diagnosis by repeatedly replaying a failure moment on the production site. However, production site application recording and replay faces deployment challenges (e.g., requiring sensitive inputs) and cannot support online diagnosis. In comparison, PerfCompass does not require application record and replay to achieve easy deployment for production systems.

### 7.3 Performance Bug Diagnosis

Our performance bug diagnosis work is closely related to Triage [112] since both provide onsite production-run failure diagnosis framework. The key idea of Triage is to use checkpoint-replay
with input/environment modification to perform just-in-time problem diagnosis by comparing normal runs and failure runs. In contrast, PerfScope does not require repeated replays, which makes PerfScope less intrusive to the production computing environment. Moreover, PerfScope specifically targets the performance bugs, which often do not manifest as code block changes between normal and failure runs. By detecting system call invocation changes and leveraging the robust function signatures, we can identify the buggy functions more effectively for performance bugs.

X-ray [22] provides automatic diagnosis support for performance anomalies in production software. It uses binary instrumentation to track the information flow in order to estimate the cost and the likelihood that a block was executed due to each potential root cause (e.g., configuration option). However, X-ray cannot diagnose the performance anomalies caused by software bugs. Similar to Triage, X-ray also requires replay support in order to offload the heavyweight binary instrumentation and analysis outside the production system. In contrast, PerfScope does not require any record and replay to identify the buggy functions.

Researchers also proposed white-box approach to analyzing performance bugs. Jin et al. [58] present a study of 109 real world performance bugs in 5 different systems. By manually checking the patches of known problems, they are able to then build an efficiency rule-based checker which was able to identify previously unknown performance problems in deployed software. However, the white-box approaches are only suitable for offline debugging due to the source code requirements. Additionally, existing white-box analysis tools mainly focus on detecting possible performance bugs instead of diagnosing a specific production-run performance bug. Directly applying existing bug detectors will lead to large false positives and false negatives as demonstrated by previous work [60].

Magpie [26] instruments middleware and packet transfer points to record fine-grained system events and correlates these events using an application specific event schema to capture the control flow and resource consumption of each request. DARC [111] identifies functions that are the main contributors to the latency peaks in a OSProf profile. Although both systems can be used to diagnose performance problems, they require instrumentations to the application or the middleware, which is difficult to be deployed in the production environment. In contrast, PerfScope does not require any application or middleware instrumentations, which makes it practical for production computing environments.

Fournier et al. [49] proposed to analyze the blocking behavior using kernel-level system events for diagnosing performance problems in multi-core systems. Similar to PerfScope, their approach only relies on the kernel level tracing, which imparts little overhead to the application. However, their approach cannot identify bug related application functions that can directly help the developer diagnose the performance bugs.

Previous work also explored black-box performance debugging schemes. Cohen et al. [38]
proposed to use machine learning models to correlate the system-level metrics with the performance anomaly states for system metric attribution. Although black-box approaches are often light-weight and non-intrusive, they can only provide coarse-grained diagnosis (e.g., identifying fault related system metrics). PBI [21] used hardware performance counters along with pre-defined predicates to characterize concurrency and semantic bugs for compiled programs. In contrast, PerfScope focuses on performance bugs and can identify bug related functions for both compiled and interpreted programs.

7.4 Static Analysis Techniques

Previous work has developed techniques to identify different bugs in applications using static rule or heuristic-based identification techniques. The most well known and widely used rule-based detector is Findbugs [23]. Findbugs works by scanning source code at compile time using pre-defined rules in order to locate easily identified common problems capable of causing runtime issues. It is difficult to apply these type of static, rule-based detectors to identify the types of bugs we examined in our study.

Work has been done to automatically identify and fix concurrency bugs using static analysis. Wu et al. [125] identify and prevent data races using user defined execution filters. Similarly, Vaziri et al. [115] identify and prevent data races using user defined atomic sets along with dataflow analysis. Upadhyaya et al. [114] use static source analysis to identify atomicity violations. Lin et al. [72] use constraint solving to automatically identify and prevent deadlocks. Wang et al. [119] use Petri Nets to prevent deadlocks. Weeratunge et al. [121] use static analysis lock-order graphs and collected profiles to ensure applications are free of deadlocks. Chew et al. [36] use static source code analysis to identify areas of code with potential atomicity problems. Each work uses pre-defined heuristics/rules to specifically target certain types of concurrency bugs. Although these techniques are effective at identifying and fixing concurrency bugs, these types of rules cannot be used to automatically identify or fix many of the bugs we have studied.

Static analysis techniques have been used to automatically identify and fix failing executions. Xiong et al. [126] use constraint solvers to identify ranges of fixes for configuration errors causing failing executions. Ocariza et al. [86] use backwards slicing to automatically suggest fixes to Javascript errors. These works all rely static analysis to identify ways to bypass concrete failure points (e.g., exception thrown). Applying these techniques to the hang bugs we studied is difficult as identifying a concrete failure point is challenging for the majority of the bugs we studied.
7.5 Dynamic Analysis Techniques

Dynamic analysis techniques have been used to identify and fix performance bugs. X-ray [22] uses symbolic execution to automatically identify and suggest fixes to performance bugs caused by configuration or input-based problems. Similarly, Strider [120] uses state-based analysis of a known configuration error to identify the likely configuration source of that error. These approaches work well when a configuration error or error input is the source of a problem. However, in many of the bugs we studied the root cause of the problem is not one of these two items. Instead, the root cause comes from other sources (e.g., unexpected component interactions, unexpected return value).

Work has also been done to automatically identify and fix concurrency bugs [66, 69, 94, 116, 134] using dynamic analysis techniques. Similar to static analysis, these works all rely on concrete rules for fixing a concurrency bug once it has been identified. As described in Chapter 7, identifying these types of simple rules is difficult for the bugs we studied. Work has also been done to prevent failing executions using dynamic program analysis [24, 74, 100]. As before, the hang bugs we studied do not have a concrete failure point, making it difficult to apply these works to the bugs we studied.

7.6 Hybrid Analysis Techniques

Some tools use a combination of static analysis and dynamic identification techniques to find and fix concurrency bugs [59, 61, 73, 107]. As discussed earlier, the types of rules required for these works cannot be used to automatically identify or fix many of the bugs we have studied. Similarly, hybrid techniques have been used to automatically fix crashing bugs [31]. These works rely on a concrete failure point to work. As mentioned earlier, identifying a concrete failure point (e.g., exception thrown) is challenging for the majority of the bugs we studied. This makes it difficult to use these existing techniques on the bugs we studied.
Chapter 8

Conclusions and Future Work

This dissertation focuses on developing a framework to handle production run performance anomalies in the cloud, a task which is critical to the future success of cloud-based technology. Our prototype framework consists of four main components: a performance anomaly prediction module, a fine-grained fault localization module, a faulty function identification module, and a hybrid static-runtime analysis diagnosis module.

The rest of this final chapter is organized as follows. We first summarize the contributions of this dissertation and we then discuss future research directions.

8.1 Conclusions

We have shown that performance anomaly prediction and diagnosis is possible using easy to collect system metrics and system calls in production cloud environments. In addition, the contributions of this study have made the initial steps in developing a framework for proactively handling performance anomalies in production-level environments. We believe this framework is the first step in enabling automated performance anomaly prediction and diagnosis in production cloud environments. Specifically, we have:

- We have designed and developed UBL, a novel black-box unsupervised behavior learning and anomaly prediction system for IaaS clouds. UBL is able to capture emergent system behavior and predict unknown anomalies by monitoring black-box metrics (e.g., CPU usage, memory usage) using Self-Organizing Maps. Our results show that UBL is able to predict performance anomalies with 5.9-87.7% higher true positive rates and 3.3-84.5% lower false alarm rates compared to similar approaches while also achieving up to 47 seconds of lead time, allowing for automated anomaly prevention.

- We have designed and developed PerfCompass, a runtime performance anomaly fault
localization tool using online system call trace analysis techniques. PerfCompass can identify whether a production-run performance anomaly is caused by an external fault (e.g., interference from other co-located applications) or an internal fault (e.g., software bug) without requiring any application source code or runtime application instrumentation. Our results show that PerfCompass is able to accurately localize all 21 of the real-world faults we tested while imposing an average of 2.1% overhead during normal application execution time.

- We have designed and developed PerfScope, a practical online performance bug inference tool designed to help developers diagnose production-run performance anomalies caused by internal software bugs. Our experiments show that PerfScope is able to present developers with a small small percentage (0.001-2.3%) of functions to examine while also successfully identifying root cause related functions. Furthermore, PerfScope does not require source code or expensive online instrumentation and imparts 0.8-3.3% runtime overhead to the production system during normal execution time.

- We have designed and developed HSR, a fine-grained bug diagnosis tool which uses runtime information along with static analysis to provide performance anomaly diagnosis. Our experiments conducted on 9 real performance bugs show that combining runtime information with static analysis can significantly reduce the number of false positives reported by up to 98% compared to static approaches and up to 91% compared to runtime approaches.

8.2 Future Directions

As the cloud grows more complex in the coming years, performance anomaly management will become a necessity. This dissertation has taken the initial steps in providing a unified performance anomaly management solution designed to work in production cloud environments. However, there is still much work to be done.

- **Concurrent faults diagnosis.** Each external environment issue or internal software bug we have used to evaluate PerfCompass was triggered in isolation. It will be interesting to study the impact of concurrent faults. We have conducted a set of initial experiments to understand the behavior of PerfCompass under concurrent faults. First, we ran an experiment to examine the combined effect of an external CPU cap fault and an internal infinite loop bug. We found that 100% of the threads were affected, causing PerfCompass to first localize the issue to be external and thereby trigger external fault handling mechanisms (e.g., live VM migration). Additionally, we found that CPU related system
calls were ranked high which gives guidance on where to migrate to. After migration
is complete and the external issue is no longer present, the infinite loop bug is then
correctly localized as internal. Second, we ran an experiment to evaluate PerfCompass
using concurrent CPU and memory contention faults. Specifically, we triggered an overly
low CPU fault and an overly low memory fault concurrently. PerfCompass calculates a
fault impact factor of 100%, correctly indicating an external issue. Also, we found system
calls related to memory were ranked the highest because the memory cap fault has the
dominating impact in this case. More detailed experiments can give additional insight
into how the system behaves under concurrent faults.

- **Distributed system bug diagnosis.** Our current evaluation is limited to single node
  server setup. For performance bugs in distributed systems, we believe that PerfScope can
help with those bugs by identifying the affected functions on each affected component.
In addition, performance bugs in distributed systems can also occur as the result of
poor interactions between different system components (e.g., incorrect timeout value).
Previous work [80] has shown that those bugs also exhibit anomalous application to kernel
interactions on the faulty components in the form of either I/O or locking. Studying how to
apply PerfScope to handle these distributed system performance bugs would be interesting
future work.

- **Automated rule extraction.** Our current implementation of HSR relies on the manual
creation of rules through careful analysis of existing bugs and their corresponding patches.
Although this standard, ad hoc approach works well, it is a time consuming task to
manually create each rule. We believe this process can be improved by automatically
inferring rules. For example, genetic programming [90] has been used to automatically
infer if then else rules for various well-known University of California at Irvine machine
learning benchmark datasets [113]. We believe these techniques could be adapted to auto-
matically identify and extract interesting statements existing across multiple documents.
These statements could then be compiled to bitcode or bytecode to extract candidate
identification rules.

Performance anomalies are a serious issue facing the cloud today and the initial work done here
by no means makes this a solved problem. A broader challenge researchers will need to address
is how to perform research at scale. That is, how to research complex problems which only occur
in large-scale production-level deployments. To do this, researchers will need to simulate large
complex cloud environments in order to evaluate any tools and techniques developed. Unfort-
unately, simulation cannot replicate the real large-scale cloud platforms with 100% accuracy.
Thus, researchers developing the next generation of performance anomaly management tools
will need to develop new simulation techniques capable of accurately replicating the behaviour of existing large-scale platforms.
REFERENCES


115


