ABSTRACT

DAI, TING. A Hybrid Approach to Cloud System Performance Bug Detection, Diagnosis and Fix. (Under the direction of Xiaohui (Helen) Gu.)

Server applications running inside production cloud infrastructures are prone to various performance problems (e.g., software hang, performance slowdown). When those problems occur, developers often have little clue to diagnose those problems. First, we present Hytrace, a novel hybrid approach to diagnosing performance problems in production cloud infrastructures. Hytrace combines rule-based static analysis and runtime inference techniques to achieve higher bug localization accuracy than pure-static and pure-dynamic approaches for performance bugs. Hytrace does not require source code and can be applied to both compiled and interpreted programs such as C/C++ and Java. We conduct experiments using real performance bugs from seven commonly used server applications in production cloud infrastructures. The results show that our approach can significantly improve the performance bug diagnosis accuracy compared to existing diagnosis techniques.

Many performance problems happen in cloud server systems such as Hadoop and Cassandra are caused by data corruption. Software hang bugs make the system become unavailable to either part of or all of the users, which is one of the most severe performance problems production systems try to avoid. Generic approaches including Hytrace cannot detect the data-corruption induced software hang bugs. Thus, we present DScope, a tool that statically detects data-corruption related software hang bugs in cloud server systems. DScope statically analyzes I/O operations and loops in a software package, and identifies loops whose exit conditions can be affected by I/O operations through returned data, returned error code, or I/O exception handling. After identifying those loops which are prone to hang problems under data corruption, DScope conducts loop bound and loop stride analysis to prune out false positives. We have implemented DScope and evaluated it using 9 common cloud server systems. Our results show that DScope can detect 42 real software hang bugs including 29 newly discovered software hang bugs. In contrast, existing bug detection tools miss detecting most of those bugs.

To correct those performance problems, especially the software hang bugs caused by data processing or inter-process communication failures in the cloud, we present an automatic fixing tool, HangFix. HangFix extracts commonly seen hang bug patterns and provides different patching strategies based on the identified bug patterns. HangFix is application-agnostic, which does not require any application source code or knowledge to produce the hang bug fix. We have implemented a prototype of HangFix and evaluated the system on 42 real-world software hang bugs in 10 commonly used cloud server applications. Our results show that HangFix can successfully fix 40 out of 42 hang bugs in seconds.

A Hybrid Approach to Cloud System Performance Bug Detection, Diagnosis and Fix

by
Ting Dai

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

2019

APPROVED BY:

Kathryn Stolee	Guoliang Jin
Matthias Stallmann	Xiaohui (Helen) Gu Chair of Advisory Committee

DEDICATION

To my family and friends.

BIOGRAPHY

Ting Dai received his Bachelor degree in 2011 and Master degree in 2014 from Nanjing University of Posts and Telecommunications. At North Carolina State University, his research is motivated by the prevalence of cloud systems and performance problems emerged on the application services running inside cloud. His research has focused on applying static code analysis and machine learning based dynamic analysis on cloud computing and system reliability. He has interned at InsightFinder and IBM Research in the summer of 2016 and 2018, respectively.

ACKNOWLEDGEMENTS

This dissertation cannot be done without the encouragement, feedback, and support of several people, whom I would like to acknowledge now.

First and foremost, I would like to thank my advisor Dr. Xiaohui (Helen) Gu. Her passion about research, knowledge of cloud computing, and experience of industry pushed my limits and made me work much harder than I ever possibly could over the past several years. I truly enjoyed working with her and it is such an experience.

I would like to thank Dr. Shan Lu for the guidance and advise she gave me in our cooperated projects. Her experience of static analysis and software bugs guided me to the right direction in my research.

I would next like to thank my dissertation committee members, Dr. Kathryn Stolee, Dr. Guoliang Jin, and Dr. Matthias Stallmann, for their insight and feedback on my dissertation.

I would also like to thank several current and former labmates, Daniel Dean, Peipei Wang, Rui Shu, Jingzhu He, Fogo Tunde-Onadele, Yuhang Lin, Shuai Yang, Anwesha Das for their precious friendship and help.

I would like to thank the National Science Foundation and Credit Suisse for the funding they have provided which made my research possible.

Last but not the least, I would like to give my special thanks to my family, my roommate and my dog for their encouragement and emotional support, without whom this would have been much more difficult.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

1.1 Motivation

Cloud computing infrastructures [App; Ec2] have become increasingly popular by allowing users to access computing resources in a cost-effective way. However, performance bugs are inevitable in deployed cloud systems. Even expert programmers introduce performance bugs, which cause serious problems [Emb]. Well tested cloud products such as Hadoop and Cassandra are also affected by performance bugs [Jin12b; Dai18b; Dai18a]. Performance bugs are notoriously difficult to debug because they produce little debugging information. When performance bugs happen, they severely impact cloud server performance and quality, prolonging server response time, reducing system throughput, wasting system resources, and leading to poor user experience [Nis13a].

1.2 Summary of the State of the Art

Much work has been done to use pre-defined rules to statically detect different types of performance bugs [Jin12b; Che14; Son14; Nis13b; Xia13; Kad09]. These bug-detection tools are suitable for detecting specific performance bugs. Once applied for generic performance diagnosis or other type performance bug detection, they suffer both the false positive and false negative problems.

Dynamic analysis techniques have been used to identify and fix performance bugs that are triggered in production environments [Att12; Wan03; Dea15b; Kim14; Han12; Agu03; Kas10; Xu09; Yu16]. Those approaches focus on identifying the faulty components, nodes or interactions that lead to performance problems, which are different from our work. We focus on detecting, diagnosing and fixing performance bugs in the software level (e.g., code analysis).

Hybrid techniques have been used to fix concurrency bugs [Jin11; Jin12a], and for statistical debugging [Chi09]. Those approaches often require failure points, error statements, or application instrumentations, which make it impractical for detecting, diagnosing and fixing performance bugs in production cloud environments.

1.3 Thesis Statement

The aim of this report is to understand the limitations of existing automated tools for detecting, diagnosing and fixing performance anomalies and to suggest new techniques for advancing the state of the art. The discoveries and findings made during our studies formulate the following thesis statement:

Leveraging both runtime environment and execution information on production cloud systems and offline software code analysis can enable us to achieve more efficient and effective performance bug detection, diagnosis and fix than existing schemes.

Parts of this thesis statement are present in each of our works. The applications we have studied are all commonly used in the cloud. The software bugs we have studied are those that commonly occur in the cloud. We quantify the effectiveness and efficiency of our techniques using standard measurements. For example, we use both coverage and precision to evaluate the effectiveness of our scheme. We measure the run-time overhead and performance impact on production systems to justify the efficiency. Other effectiveness and efficiency metrics can be found in each of our studies.

We first develop a hybrid approach to pinpoint the root cause functions of a performance anomaly in cloud systems. We then propose to conduct static analysis to detect data corruption related infinite loops in application functions which cause software hang in cloud systems. Finally, we present a fixing tool which can automatically correct a software hang bug caused by data processing or inter-process communication failures in the cloud.

1.3.1 Research Challenges

Providing a practical, efficient, and effective solution for detecting, diagnosing and fixing performance bugs in cloud systems requires overcoming following challenges:

- **Agile operation:** Due to the severe financial cost performance anomalies cause, it is desirable to predict and prevent a problem before it gets triggered in production. If this is not possible, identifying the root cause of the problem and proposing a correct fix for the problem as quick as possible is a top priority. As a result, our approaches should provide results in a matter of seconds or minutes as opposed to hours or days.

- **Application agnostic:** Cloud server applications have different implementations due to different purposes. Therefore, we need application agnostic approach that requires no system-specific knowledge.

- **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 report, we make the following contributions:

- We present Hytrace, a hybrid approach to diagnosing real-world performance bugs in production cloud systems. Hytrace combines rule based static analysis and runtime inference techniques to achieve higher accuracy than pure-static or pure-dynamic approaches. Hytrace does not require any application source code or instrumentation, which makes it practical for production cloud environments. We have implemented a prototype of Hytrace and tested it over 133 real performance bugs discovered in different commonly used server applications. Our results show that Hytrace can greatly improve coverage and precision comparing with existing state-of-the-art techniques. Hytrace is light-weight, which imposes less than 3% CPU overhead to the testing cloud environments.

- We present DScope, a new data corruption hang bug detection tool for cloud server systems. DScope combines candidate bug discovery and false positive pattern filtering to detect software hang bugs that are related to data corruptions. DScope is fully automatic without requiring any user input or pre-defined rules. We have implemented a prototype of DScope and evaluated it over 9 commonly used cloud server systems. DScope successfully detects 42 true corruption hang bugs (29 of them are new bugs) while existing bug detection tools can only detect very few of them (2 by Findbugs and 1 by Infer).

- We present HangFix, an application-agnostic hang bug fixing tool for automatically recovering a cloud service outage caused by unexpected data processing or inter-process communication failures. HangFix identifies different hang bug patterns and produces corresponding software patches based on automatically generated patching strategies using intra-/inter-procedural analysis and data-dependency analysis. We have implemented a prototype of HangFix and evaluated it on 42 real-world software hang bugs in 10 commonly used cloud server systems. HangFix successfully fixes 40 out of 42 hang bugs within seconds.

This report is organized as follows. Chapter 2 describes a hybrid approach to diagnosing real-world performance bugs in production cloud systems. Chapter 3 presents a new data corruption hang bug detection tool for cloud server systems. Chapter 4 talks about an automatic software hang bug fixing tool for cloud systems. Chapter 5 compares our work with related work. Finally, Chapter 6 concludes our discussion of detecting, diagnosing and fixing performance bugs and describes future directions for our work.

CHAPTER

2

HYTRACE: A HYBRID APPROACH TO PERFORMANCE BUG DIAGNOSIS IN PRODUCTION CLOUD INFRASTRUCTURES

2.1 Introduction

When a performance problem (e.g., software hang, performance slowdown) occurs in production cloud infrastructures, it is notoriously difficult to diagnose because the developer often has little diagnostic information (e.g., no error log or core dump) to localize the fault. A recent study [Jin12b] has also shown that performance bugs widely exist across different server applications that are commonly used in production cloud environments.

Previous work on performance bugs can be broadly classified into two groups: 1) *static analysis schemes* [Fbi; Fin; Jin12b; Nis15] that detect bugs by searching specific performance anti-patterns in software, such as inefficient call sequences or loop patterns; and 2) *dynamic runtime analysis schemes* [Aru13; Dea14] that closely monitor runtime application behaviors to infer root causes of performance problems.

Both approaches have advantages but also limitations. The static analysis approach imposes no runtime overhead to production systems. However, without run-time information and without focusing on the specific anomaly occurred in a production run, this approach inevitably suffers from

excessive false alarms, reporting code regions that are unrelated to the production run performance problem. To address this problem, previous work proposed specialized rule checkers to detect specific and known performance bugs [Jin12b; Nis15]. However, specialized rule checkers cannot cover many real world performance bugs as shown in our experiments.

In contrast, a dynamic approach can target the specific problem that has occurred in the production environment. However, it needs to perform monitoring on production systems, inevitably imposing overhead. To avoid excessive runtime overhead, previous research proposed performance diagnosis based on system-level metrics or events that can be easily collected with low overhead, such as CPU utilization, free memory, system calls, and performance-counter events [Aru13; Dea14]. Unfortunately, without knowledge about program semantics, those dynamic techniques suffer from both false positives and false negatives too [Aru13; Dea14].

2.1.1 A Performance Bug Example

To illustrate the challenge of performance bug diagnosis, we discuss Apache-37680[1] bug. This bug was discovered when a user conducted a graceful restart to Apache server, after he/she modified Apache configuration, changing the web server from listening to two ports to just one port. The graceful restart option attempts to minimize any downtime by only restarting parts of the application. However, instead of coming back online in a few seconds as expected, Apache server hangs, consuming 100% CPU in the process.

The direct cause of this problem is a blocking call Apache attempts to make on the single port during graceful restart. Since the configuration of the socket does not allow blocking calls, Apache endlessly re-tries the call and hangs. The root cause is related to the (un)blocking setting of the port. In this bug, graceful restart reuses the socket from the previous running instance, without changing the socket setting. Unfortunately, this socket was set to not allow blocking calls in `ap_setup_listeners` function through the `apr_socket_opt_set` function in previous running instance, when two ports were configured.

The patch only makes one major change, as shown in Figure 2.1. Instead of a constant value "1", a variable `use_nonblock` is passed to the invocation of `apr_socket_opt_set` inside the function `ap_setup_listeners`. This variable controls whether the socket is configured to allow or not allow blocking calls. This change allows graceful restarts to enable blocking calls on the reused socket before making blocking calls.

It is challenging to precisely detect the above problem using pure static checking. A rule that precisely captures the root cause of this bug is that blocking calls should not be made on a socket configured to not allow blocking calls. This rule is almost infeasible to check statically — the socket configuration can happen long before the blocking call, and inter-procedural path-sensitive static analysis cannot scale to complicated production server software. Furthermore, even if this rule is checkable, it is too specific. Providing good diagnosis coverage using such specific rule checking is difficult if not totally impossible. Note that, a traditional generic infinite loop detector would not

[1]We use "application name *dash* bug identifier" in the repository to denote each bug in this thesis.

```
//listen.c                                                    Apache-37680(v2.0.55)
479 - if (ap_listeners && ap_listeners->next) {
    + use_nonblock = ap_listeners && ap_listeners->next;
480     for (lr = ap_listeners; lr; lr = lr->next) {
481       apr_status_t status;
          ...
483       status = apr_socket_opt_set(lr->sd, APR_SO_NONBLOCK,
484 -                                                        1);
    +                                                   use_nonblock);
485       if (status != APR_SUCCESS) {
486           ap_log_perror(APLOG_MARK, APLOG_STARTUP | APLOG_ERR, status, pool,
487                                                    "ap_listen_open: unable to "
488 -                                                  + "make socket non-blocking");
    +                                        + "control socket non-blocking status");
489         return -1;
490       }
491     }
492 - }
```

Figure 2.1 The patch for Apache-37680 bug. The patch is inside function `ap_setup_listeners`. The bug occurs as a result of the constant value "1" being passed to the `apr_socket_opt_set` function, causing an infinite loop in another function at runtime. "+" means the added lines while"-" means the deleted lines, representing the patch for the bug.

work here, because it cannot reason about the fact that a blocking call will always fail on a socket under certain configuration.

It is also challenging to precisely diagnose the above problem using purely dynamic techniques. Dynamic techniques often try to discover (statistically) abnormal execution behaviors based on traces of system calls [Dea12; Dea14], performance counters [Aru13], or other system metrics [Mys14; She09]. Unfortunately, for bugs like the one in Figure 2.1, the above dynamic techniques will discover the symptom but unable to discover the root cause, which does not produce abnormal system-call or performance-counter features. Furthermore, these techniques tend to introduce false alarms due to the inherent uncertainty nature of the statistical behavior modeling. Finally, without source code access, it could be nontrivial for developers to associate dynamic diagnosis results with specific source-code level buggy functions or buggy lines.

2.1.2 Our Contribution

This chapter presents Hytrace, a novel *hybrid* performance bug diagnosis scheme for production cloud infrastructures. Our technique does not require any application source code and imposes little overhead, which makes it practical for the production cloud environment. Hytrace achieves both higher *coverage* and better *precision* than existing pure-static and pure-dynamic schemes.

The key challenge in designing such a hybrid scheme is to retain the strengths and alleviate the weakness of each individual scheme. Our idea is to construct a static anti-pattern detector and a dynamic abnormal behavior detector that each individually provides *high coverage* maybe at the expense of precision. When combining such schemes, the high coverage will naturally be retained and the lost precision fortunately can be regained as most false alarms would not be reported by both schemes that conduct diagnosis from different perspectives.

Specifically, we propose a generic rule checker that statically detects functions that bear code patterns vulnerable to potential performance problems. When a performance problem such as hang or slowdown is observed by users or automated monitors [Bha08; Kut10; Ngu13a; Tan12], we use run-time analysis to identify a ranked list of functions that produce abnormal system-level metrics during the production run either themselves or through their immediate callees. Functions that appear suspicious from both static and dynamic analysis are reported.

Intuitively, our static scheme captures performance-bug-prone code patterns while our dynamic scheme captures abnormal runtime behaviors. The combination of the two leverages both program semantic and run-time behavior information, and hence can achieve higher precision than pure-static or pure-dynamic techniques.

This chapter makes the following contributions:

- Hytrace — We present Hytrace, a novel hybrid performance diagnosis approach that combines runtime inference with static analysis to achieve a better combination of accuracy, coverage, and efficiency in performance anomaly diagnosis than existing schemes.

- Hytrace-static — We develop a rule-based static analysis tool that can detect potential performance problems in server applications. This tool aims at achieving higher detection rate than existing static analysis tools. For generality, our tool strives to support both C/C++ and Java.

- Hytrace-dynamic — Hytrace leverages and extends an existing dynamic analysis tool [Dea14] to conduct low-overhead run-time performance anomaly inference with higher diagnosis coverage than existing pure-dynamic analysis schemes.

- We implement Hytrace and evaluate it using 133 real performance bugs (14 of them are reproduced by us) in seven commonly used server applications (Apache, MySQL, Lighttpd, Memcached, Hadoop, Cassandra, Tomcat) reported by production cloud users.

Note that, Hytrace framework is extensible and configurable: we can add new rules or drop existing rules to and from Hytrace-static module easily, and we can replace Hytrace-dynamic module with any other runtime analysis tools as long as those tools follow our design principles (e.g., low overhead, high coverage).

Our results show that Hytrace significantly improves the *accuracy*, with the true root-cause' ranking improved from top 10 to top 3 (on average) for diagnosing 14 *reproduced* performance bugs compared to existing pure-dynamic analysis tools (PerfScope [Dea14]). None of these bugs can be covered by traditional pure static checkers that target on general software bugs (Infer [Fbi], Findbugs [Fin]) or specific types of loop inefficiency bugs (Caramel [Nis15]). Moreover, Hytrace-static improves the *coverage* by at least 69% for diagnosing 133 performance bugs compared to Infer, Findbugs and Caramel. Hytrace is light-weight: imposing less than 3% overhead to the systems and localizing suspicious functions for complex server applications with millions lines of code within tens of minutes.

2.2 Design

This section first describes our static analysis and dynamic analysis components separately, and then describes how their results are combined.

2.2.1 Hytrace Static Analysis

Our static analysis module focuses on detecting potential faulty functions that are prone to performance problems. Its design includes two parts. First, design the target for static analysis — identify a few static code patterns that are vulnerable to performance problems, which we will refer to as *rules*. Second, design the static analysis algorithm — design how to analyze the program and discover code regions that match those rules.

Rule Design Principles Our rule design follows two principles. First, different from many standalone static checkers, our design favors generality over precision. We should look for code patterns that are maybe-indicators of performance problems, not patterns that are guaranteed to cause performance problems. This principle helps us avoid missing true buggy functions. Since the runtime inference component of Hytrace can effectively filter out many falsely identified functions detected by the static analysis, the final precision of Hytrace will be much better than the precisions of these static rules.

Second, like that in all static checkers, we should find statically checkable rules. That is, whether a code region matches a rule or not should be decidable without any runtime information. For example, checking whether a function call uses a constant value as a parameter is statically checkable. In contrast, whether a variable can take on a particular value during program execution often cannot be checked statically.

We randomly sampled 20 out of 133 performance bugs. We have derived a set of rules that meet our design principles empirically based on our experience of studying those 20 real-world performance bugs in server applications. For the purpose of cross validation, we use another disjoint set of 20 bugs to perform the same rule extraction process. (the details about all the 133 bugs are available online [Hyt]). We found that we extract the same set of generic performance bug detection rules. Those 40 sample bugs are our rule generation training set. The 133 bugs form Hytrace testing set and are used in our experimental evaluation. Note that, Hytrace can be easily extended with other rules that follow our design principles and integrated with any static analysis tools that can identify a set of candidate performance-problem-prone functions. We now describe the rules used by Hytrace static analysis component in detail as follows.

R1: Constant parameter function calls. A function call that uses a constant value as a primitive-type parameter matches this rule; the function that issues such a constant-parameter function call will be considered as a candidate faulty function. Clearly, this rule is generic, not limited to any specific software, and statically checkable. Furthermore, it does reflect a common performance problem — hard-coded parameters cannot handle unexpected workload, configuration, or environment. For example, the Apache bug discussed in the introduction uses a constant parameter in function

```
//sql_insert.cc                                                    MySQL-28000(v5.0.37)
1260  if (fill_record_n_invoke_before_triggers (    thd, *info->update_fields,
1261                                                    *info->update_values,
1262 -                                                                    0,
     +                                                                info->ignore,
1263                                          table->triggers, TRG_EVENT_UPDATE))
```

Figure 2.2 Example for R1: constant parameter (MySQL-28000 bug). The bug occurs as a result of the constant value 0 being passed to the invocation of `fill_record_n_invoke_before_triggers` in function `write_record`, causing an endless loop at runtime. "+" means the added lines while"-" means the deleted lines, representing the patch for the bug.

```
//RMContainerRequestor.java                                        Mapreduce-5489(v2.2.0)
148  AllocateRequest allocateRequest = AllocateRequest.newInstance(
149                                  lastResponseID, super.getApplicationProgress(),
150      new ArrayList<ResourceRequest>(ask), new ArrayList<ContainerId>(release),
151 -                                                                        null);
    +                                                                  blacklistReq);
```

Figure 2.3 Example for R2: null parameter (Mapreduce-5489 bug). The bug occurs as a result of not using node blacklisting feature in Resource-Manager requests, hanging MapReduce jobs. "+" means the added lines while"-" means the deleted lines, representing the patch for the bug.

`apr_socket_opt_set`, which makes the socket only support non-blocking calls. This predefined functionality cannot handle unexpected configuration changes (i.e., changing the number of the listening ports from 2 to 1). As another example, the MySQL-28000 bug shown by Figure 2.2 uses a hard-coded constant value of 0, which causes MySQL to never ignore errors when executing the `fill_record_n_invoke_before_triggers` function. In most cases, this is not a problem as errors would be handled appropriately (e.g., logged). However, in certain circumstances, such as, when executing the `INSERT IGNORE` command, errors should be ignored but are not, which causes the system to hang.

Many performance problems are related to function calls that match this rule, yet matching this rule does not mean performance problems will necessarily happen. This matches our design principle.

R2: Null parameter function calls. A function call that uses `null` as a pointer/object parameter matches this rule; the function that issues such a null-parameter function call will be considered as a candidate faulty function. This type of function calls can be related to performance problems in several ways. The `null` parameter is sometimes unexpected and hence not properly handled, leading to unexpected execution behavior. Sometimes, the `null` is the default parameter. When developers "lazily" use a default parameter, inefficiency may follow. Figure 2.3 shows an example for this rule. In this bug, `null` is the default value for the last parameter of `AllocateRequest::newInstance` method. Using the default value causes the Resource-Manager to not blacklist any bad Node-Managers during job allocation, even when a bad Node-Manager is already blacklisted by the Application-Master. As a result, the Resource-Manager could keep allocating the same blacklisted Node-Manager to the Application-Master, leading to a hang problem.

R3: Unsafe function calls. Some widely used I/O library functions, such as `atol` and `fopen`, may return unexpected output. When those unexpected return values are not properly handled, the

```
//mod_proxy_http.c                                          Apache-40883(v2.2.3)
336 -   cl_val = atol(old_cl_val);
    +   if (APR_SUCCESS != (status = apr_strtoff(&cl_val, old_cl_val, NULL, 0))) {
    +     return status;
    +   }
        ...
340     while (!APR_BUCKET_IS_EOS(...input_brigade)){
         ...
343      bytes_streamed += bytes;
         ...
363      if (bytes_streamed > cl_val)
364        continue;
         ...
         //input_brigade is changed after
409    }
```

Figure 2.4 Example for R3: unsafe function (Apache-40883 bug). The bug occurs as a result of calling `atol` to convert a string, whose value is greater than 2 GB, into a long integer in `stream_reqbody_cl` function, hanging Apache system. "+" means the added lines while"-" means the deleted lines, representing the patch for the bug.

```
//mod_deflate.c                                             Apache-51590(v2.4.1)
452   apr_bucket *e = APR_BRIGADE_FIRST(bb);
453   while (1) {
          ...
457       if (APR_BUCKET_IS_EOS(e)) {
458           ap_remove_output_filter(f);
459           return ap_pass_brigade(f->next, bb);
460       }
461       if (APR_BUCKET_IS_METADATA(e)) {
    +         e = APR_BUCKET_NEXT(e);
462           continue;
463       }
          ...
471   }
```

Figure 2.5 Example for R4: unchanged loop exit condition variables (Apache-51590 bug). The bug occurs as the highlighted while loop becomes an infinite loop due to wrong handling along the `APR_BUCKET_IS_METADATA` branch, hanging Apache system. "+" means the added lines, representing the patch for the bug.

affected system may hang. We define those functions as unsafe functions. This rule checks whether an unsafe function is called and reports the function that calls an unsafe function as a candidate faulty function. For example, Figure 2.4 shows the patch for Apache-40883 bug. In this bug, an unsafe function `atol` is called by `stream_reqbody_cl` to convert the `old_cl_val` string into the long integer, `cl_val`. This string happened to be larger than 2GB on the user's 32-bit machine. The integer overflow caused `atol` to return 0, which in turn caused the if branch be taken in every iteration of the while loop. Once that happens, a `continue` statement is executed without updating `input_brigade` and then goes to the loop header, the same `input_brigade` value makes the condition of while loop always be true, causing whole Apache to hang. The patch simply replaced `atol` with its large-file alternate: `apr_strtoff`.

R4: Unchanged loop exit condition variables. This rule looks for the loops whose exit condition

```
//connections.c                                          Lighttpd-2197(v1.4.27)
238     if (len < 0) {
          ...
307       return -1;
308     } else if (len == 0) {
          ...
312       return -2;
    +   } else {
    +     joblist_append(srv, con);
313     }
314     return 0;
```

Figure 2.6 Example for Rule 5: uncovered branch (Lighttpd-2197 bug). The bug occurs as a result of unhandling fragmented ssl request case, stalling Lighttpd. "+" means the added lines, representing the patch for the bug.

variables should be updated but not changed by mistake and reports the function that contains such a loop as a candidate faulty function. The rationale behind the rule is that an infinite loop can occur when the exit condition variables are unchanged, which may cause software hang performance problems. Figure 2.5 shows the patch for Apache-51590 bug. In this bug, a `while` loop is called by function `deflate_out_filter` when reading buckets. When the input brigade contains a metadata bucket, the second `if` branch will be taken. Once that happens, a `continue` statement is executed without moving the pointer `e`. The pointer `e` is a loop exit condition related variable. After that, in each iteration, function `deflate_out_filter` processes the same metadata bucket without moving the pointer `e` and then goes back to the loop header, i.e., `while(1)`, causing a hang. The patch updates the loop exit variable `e` by adding a statement to move the pointer to the next bucket in the `APR_BUCKET_IS_METADATA` branch, making sure that loop does not stuck at a metadata bucket.

R5: Uncovered branch. A function which does not cover all branches of conditional statements matches this rule. Those uncovered cases might be poorly handled, leading to unexpected execution behavior. Figure 2.6 shows the patch for Lighttpd-2197 bug. Function `connection_handle_read_ssl` did not handle the `len > 0` branch. When ssl requests are sent in multiple fragments (i.e., `len` is positive), `connection_handle_read_ssl` just drops the fragmented packages silently, which in turn stalls Lighttpd system and causes frequent timeouts at client ends. The patch simply added the `else` branch to push the fragmented package into `joblist`.

Rule-Checking Analysis We develop static checkers to find suspicious functions that match the above rules. We choose to analyze intermediate representation (e.g., LLVM bitcode) or object code (e.g., Java bytecode) as opposed to source code, which allows Hytrace to work in production cloud infrastructures where applications often belong to the third-party and the source code is often unavailable. Of course, not operating at the source level has its challenges. For example, in Java, function calls are converted into the `invokedynamic` instruction, with n arguments being the previous n instructions before it. We need tools that can correctly extract arguments. We have developed several extensible binary bug checkers using existing static analysis frameworks. Specifically, we use LLVM [Llv] for C/C++ applications and Findbugs [Fin] for Java applications. In

addition to providing rule-checking functionality, our checkers provide several utility functions, such as `invokedynamic` argument extraction. Additionally, Hytrace framework allows users to easily add new rules with few code changes. We will describe the implementation details in Section 2.3.

2.2.2 Hytrace Dynamic Analysis

The design principle of Hytrace dynamic analysis component is similar to that of static analysis part. Our goal is to relax the requirement for *precision* and maximize the *coverage* (i.e., avoid miss detections) by including all potential root cause related functions. The current Hytrace-dynamic module extends an existing dynamic cloud performance debugging tool PerfScope [Dea14] to achieve our design goal. We chose PerfScope because it imposes low overhead and does not require source code access, which makes it practical for production cloud infrastructures. However, like other dynamic techniques [Aru13; Att12; Han12; Kim14; Wan03], PerfScope sacrifices *coverage* in order to achieve high *precision*, which makes it inevitably miss identifying buggy functions that have major contributions to the root cause. Based on this, Hytrace-dynamic is proposed to address the *coverage* issue in PerfScope.

When a performance anomaly is detected by an existing online anomaly detection tool [Dea12; Tan12], we first trigger a runtime system call analysis to identify abnormal system call sequences produced by the server applications. Specifically, we analyze a window of recent system call trace and identifies which types of system calls (e.g., `sys_read`, `sys_futex`) experience abnormal changes in either execution frequency or execution time. We first divide a window of system call trace into multiple execution units based on the thread ID. We then apply a top-down hierarchical clustering algorithm [Kau09] to group those execution units that perform similar operations together based on the appearance vector feature. Next, we use the nearest neighbor algorithm [Tan05] to perform outlier detection within each cluster to identify abnormal execution units. Frequent episode mining [Agr94; Pat12] on those abnormal execution units is then used to identify common abnormal system call sequences (i.e., `S1`). For example, from the trace of HDFS-3318, a sequence {`sys_gettimeofday`, `sys_read`, `sys_read`, `sys_gettimeofday`} is discovered to be executed more often than usual.

Next, we identify application functions that have issued the abnormal system call sequences identified above. We again use frequent episode mining to extract common system call sequences (i.e., `S2`) produced by different application functions. These sequences (`S2`) are then used as signatures to match with system call sequences (`S1`) whose execution frequencies or time are identified to be abnormal. For example, in HDFS-3318, function `Reader.performIO` is found to often produce system call sequence {`sys_gettimeofday, sys_read, sys_read, sys_gettimeofday`}, which is then used as its signature. When we detect {`sys_gettimeofday, sys_read, sys_read, sys_gettimeofday`} as one of the abnormal system call sequences, `Reader.performIO` is matched as one candidate buggy function.

In comparison to existing dynamic analysis tools (e.g., PerfScope), Hytrace-dynamic integrates runtime execution path analysis with abnormal function detection in order to increase the bug

detection coverage. Specifically, we extend the candidate function list by adding the *k*-hop caller functions of those abnormal functions identified by the dynamic analysis tool. We also conducted sensitivity study on the number of caller function hops (e.g., *k*) to evaluate the tradeoff between coverage and precision.

We then calculate a rank score for each identified abnormal function using a maximum percentage increase metric (i.e., the largest count increase percentage among all the matched syscall sequences between S1 and S2) to quantify the abnormality degree of different abnormal functions. We rank all the identified abnormal functions using increasing rank scores. The rank of the inserted caller function inherits the rank of the callee function (i.e., the identified buggy function). If a function is called multiple times and has multiple different caller functions, we add *all* the caller functions into the final list. We currently rely on the call path information extracted runtime to identify caller functions. We can also leverage any in-situ call path extraction tool that do not require application source code and impose low overhead to the production cloud environment (e.g., [Ngu14]). If a function has multiple appearances in the final buggy function list, we only keep its highest rank.

2.2.3 Hybrid Scheme

The key idea of Hytrace is to combine static and dynamic analysis techniques for achieving both high *coverage* and high *precision* performance diagnosis.

Hytrace-static favors the *coverage* (i.e., completeness) over *precision*. It captures all the potential buggy functions who are vulnerable to the performance problems over static code pattern matching. Hytrace-dynamic favors both the *coverage* and the "relaxed" *precision*. It identifies *all* the caller functions and the buggy functions who have abnormal practices during runtime.

Hytrace approach leverages the two carefully designed static and dynamic analysis components that are complementary to each other. Although each component is prone to false positives, the combination of the two leverages both program semantic and run-time behavior information, and hence can achieve much higher precision than pure-static or pure-dynamic techniques.

When a performance symptom like a hang or a slowdown is observed by either users or an automated monitoring tool, Hytrace runs its dynamic component to identify a ranked list of functions that behave suspiciously, judging by the abnormality of system-level metrics. Hytrace then compares this list produced by the dynamic component with the list of suspicious functions identified by Hytrace static component, removes the functions that only appear in one list, and adjusts the ranks of the remaining functions accordingly. For example, if Hytrace-dynamic identifies three buggy functions foo (rank: 1), bar (rank: 2), and baz (rank: 3) but bar does not include any static anti-patterns, bar is removed and the final list becomes foo (rank: 1), baz (rank: 2). The rank of baz gets improved because we remove the false positive function bar.

Hytrace enhances the bug detection precision by pruning those false positive functions which are detected by either Hytrace-static or Hytrace-dynamic but not the both. For example, in the Apache-45856 bug, Hytrace-dynamic identifies the `connect` function as suspicious because it invokes a set

of system calls with abnormal execution time. However, `connect` does not include any static anti-patterns. Thus, `connect` is a false positive function, which is pruned by Hytrace. Another example is the Cassandra-5064 bug. Hytrace-static identifies the `extractKeysFromColumns` function as suspicious because it matches the "uncovered branch" rule. However, `extractKeysFromColumns` does not have any abnormal behavior during runtime. Thus, `extractKeysFromColumns` is a false positive function, which is pruned by Hytrace.

Hytrace can also support distributed performance bug diagnosis. We define a distributed performance bug to be a bug that causes a performance anomaly (e.g., hang, slowdown) to a distributed system with more than one node. When a performance anomaly is detected, we run Hytrace concurrently on all the nodes or a subset of faulty nodes identified by other online anomaly detection tool [Ngu13a]. We can derive a buggy function list for each faulty node. We can also present a consolidated buggy function list by taking the intersection among all the buggy function lists produced by different faulty nodes.

2.3 Implementation

To perform static analysis for C/C++ applications, we have developed code analysis passes using LLVM [Llv]. LLVM is a compiler infrastructure which allows developers to examine/modify code as it's being compiled. We have implemented our static analysis as LLVM passes through the `FunctionPass` and `LoopPass` class interfaces: the former examines every application function, and the latter identifies and examines every loop inside every function. Hytrace takes the application binary code as input and converts the binary into LLVM IR using Clang [Cla].

For Java applications, we implemented our transformations using Findbugs analysis infrastructure [Fin]. Findbugs is a tool designed to analyze Java bytecode using the Apache Byte Code Engineering Library (BCEL). Apart from a variety of built-in patterns that reflect bad coding practices, Findbugs also allows users to write custom bug detectors in the form of plugins, through which we have implemented Hytrace static analysis for Java programs. These plugins can be used directly on a target directory of Java programs or easily integrated into the build process of the whole target program.

2.4 Evaluation Methodology

Our experimental evaluation uses 133 real-world performance bugs: 53 C/C++ performance bugs from 5 server applications (Apache http web server, Lighttpd web server, Memcached distributed memory caching system, MySQL database engine, and Squid web proxy) and 80 Java performance bugs from 4 server applications (Cassandra distributed key-value store, Hadoop MapReduce distributed computing infrastructure, HDFS distributed file system, and Tomcat application server). Those bugs are collected by searching for the terms hangs, *100% CPU*, *stuck*, *slowdown* and *performance* in JIRA [Jir] and Bugzilla [Bug].

Table 2.1 Descriptions of the 14 real-world bugs we reproduced.

Bug name	Root-cause description	Symptom
Apache-37680	Make a blocking "accept" call with non-blocking configuration.	hang
Apache-43238	Set up new connections with non-`keep-alive` configuration.	slowdown
Apache-45856	Call `fopen` on file > 2 GB on 32-bit systems.	hang
Lighttpd-1212	Keep processing same event when the return value `errno` is mishandled.	hang
Lighttpd-1999	Keep reading and discarding response data while processing header information.	slowdown
Memcached-106	Keep reading a non-existent package when the previous packages overwrite the read buffer.	hang
MySQL-54332	Two threads execute the `INSERT DELAYED` statement but one of them has a locked table.	hang
MySQL-65615	5 × slowdown in the table insertions after truncating a large table.	slowdown
Cassandra-5064	`ALTER TABLE` command keeps flushing empty `Memtable`.	hang
HDFS-3318	HDFS client keeps reading a > 2 GB file when the file length is represented by an `int`.	hang
Mapreduce-3738	Endless wait for an atomic variable to be set.	hang
Tomcat-53450	Tomcat tries to upgrade a read lock to a write lock.	hang
Tomcat-53173	Keep dropping incoming requests when the `count` is improperly updated.	hang
Tomcat-42753	Keep processing the same `Comet` events on a request whose filter chain is not configured.	hang

Note that, using those keywords to search performance bugs is not an accurate but easy, fast and possibly complete way to do in practice. And in this thesis, we consider *performance bugs* as those bugs when they happen, they can waste either partial (manifested as performance degradation) or all system resources (manifested as hang). The performance bugs in our benchmark are difficult to diagnose. Even if their symptoms are hang, figuring out the root cause functions behind the hang is non-trivial.

We successfully reproduced 14 performance bugs out of the 133 bugs we studied. Those 14 bugs do not overlap with the 40 sample bugs in our rule generation training set. Reproducing real-world performance problems is extremely time-consuming, sometimes taking developers up to a whole year [Sah14], and tricky due to limited and often ambiguous information [Jin12c] in bug reports. For each of these 14 bugs, we followed the original bug report to reproduce the bug and confirm the manifestation of the corresponding performance anomaly symptoms (e.g., 100% CPU usage, unresponsive system, prolonged delay). Table 2.1 shows the 14 performance bugs that we reproduced and tested. 12 of 14 bugs follow single-node configuration and the other 2 follow two-node-cluster configuration. Among all the 133 bugs, we found 125 of them are hang bugs and only 8 are slowdown bugs. The 14 reproduced bugs in Table 2.1 follow the similar statistics. In addition, these 14 bugs include all the benchmarks in the PerfScope paper [Dea14]. Thus, we believe that the 14 reproduced bugs are representative of the 133 bugs.

The Apache, Memcached, Cassandra, HDFS, Mapreduce and Tomcat systems were tested on a private cloud in our lab where each host is equipped with a Quad-core Xeon 2.53GHz CPU along

with 8GB memory and runs 64-bit CentOS 5.3 with KVM 0.12.1.2. The Lighttpd and MySQL systems were tested on the virtual computing lab (VCL) [Vcl], 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 trace was collected in a virtual machine using the kernel system call tracing tool LTTng 2.0.1 [Des06] running 32-bit Ubuntu 12.04 kernel v3.2.0.

Our experiments use the same workloads as PerfScope [Dea14] for the 12 bugs used by PerfScope. For the newly added Apache bug, we initiated 200 threads to use `httperf` to request various pages from the Apache server for 3 minutes. For Memcached, we set up a two-node cluster and wrote a multi-threaded client to send 10 million UDP requests to the server nodes.

Our evaluation looks at both coverage (i.e., true positives) and precision (i.e., false positives) of performance bug diagnosis. We compare Hytrace with several state-of-the-art static and dynamic bug analysis tools, such as, Caramel [Nis15], Findbugs [Fin], Infer [Fbi] and PerfScope [Dea14].

Table 2.2 The coverage and precision of different schemes. "perf.": using only performance-related patterns/rules in Infer and Findbugs; "all": using all patterns/rules in Infer and Findbugs. "*": Infer identifies bug-irrelevant problems in bug-related functions; "-": not supporting applications in specific languages (Caramel and Findbugs) or runtime execution errors (Infer).

Bug name	**Hytrace**		**Hytrace-dynamic**		**Hytrace-static**		**Infer (all)**		**Infer (perf.)**		**Findbugs (all)**		**Findbugs (perf.)**		**Caramel**		**PerfScope**	
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
Apache-37680	✓	17	✓	22	✓	40013	✗	18	✗	3	-	-	-	-	✗	4	✓	14
Apache-43238	✓	6	✓	12	✓	42273	✗	18	✗	2	-	-	-	-	✗	2	✓	8
Apache-45856	✓	5	✓	10	✓	32128	✗	18	✗	2	-	-	-	-	✗	4	✓	40
Lighttpd-1212	✓	2	✓	3	✓	4705	✗	181	✗	61	-	-	-	-	✗	0	✓	0
Lighttpd-1999	✓	4	✓	4	✓	5057	✗	171	✗	56	-	-	-	-	✗	0	✓	1
Memcached-106	✓	2	✓	4	✓	3983	-	-	-	-	-	-	-	-	✗	0	✓	3
MySQL-54332	✓	6	✓	11	✓	98408	-	-	-	-	-	-	-	-	✗	22	✓	2
MySQL-65615	✓	2	✓	21	✓	99076	-	-	-	-	-	-	-	-	✗	8	✓	4
Cassandra-5064	✓	1	✓	8	✓	2982	✓*	2904	✓*	2904	✗	322	✗	24	-	-	✓	3
Mapreduce-3738	✓	7	✓	17	✓	9646	✓*	5077	✓*	5077	✗	1261	✗	170	-	-	✓	11
HDFS-3318	✓	2	✓	13	✓	10767	✗	2367	✗	2367	✗	1401	✗	168	-	-	✓	4
Tomcat-53450	✓	8	✓	24	✓	4198	✗	4638	✗	4638	✗	477	✗	53	-	-	✓	1
Tomcat-53173	✓	15	✓	53	✓	3997	✗	4624	✗	4624	✗	422	✗	51	-	-	✓	13
Tomcat-42753	✓	2	✓	12	✓	4279	-	-	-	-	✗	889	✗	238	-	-	✓	28
Avg.	100%	6	100%	15	100%	25822	14%	2002	14%	1973	0%	795	0%	117	0%	5	100%	9

2.5 Experimental Evaluation

Overall, Hytrace achieves both higher coverage and better precision than existing pure static techniques and pure dynamic techniques. We discuss these evaluation results in detail below.

2.5.1 Coverage and Precision Results

Table 2.2 shows the coverage and precision results achieved by different algorithms for the 14 real performance bugs reproduced by us. Hytrace successfully identifies bug-related functions in all cases. We manually validated that the functions discovered by Hytrace are indeed related to the performance anomaly. We will provide several examples in Section 2.5.3. In contrast, existing pure static analysis schemes achieved very low coverage. In fact, most of them fail to identify any bug related functions in the 14 real performance bugs. This is because existing static analysis schemes focus on matching unique rules of specific performance problems or bad programming practices rather than discovering all possible performance problems. For example, Caramel focuses on loops that execute unnecessary iterations, which are not the root causes for any of the 14 performance bugs shown in Table 2.2. Findbugs targets bugs that follow specific patterns in Java programs, such as "method calls static math class method on a constant value", "private method is never called", "method concatenates strings using + in a loop". None of those rules match the root cause of our tested 14 performance bugs. Infer mostly focuses on memory and resource leak bugs, especially in C programs, which are also not the root causes for the performance problems shown in Table 2.2. In contrast, Hytrace static patterns favor generality over specification, which enhances the coverage for the performance bugs in our benchmark.

Infer identifies the bug-related function `maybeSwitchMemtable` for Cassandra-5064, as it discovers that `maybeSwitchMemtable` could invoke a function returning `null`. However, the performance anomaly actually happens when a non-`null` string is returned. Infer identifies the bug-related function `AppLogAggregatorImpl.run` for Mapreduce-3738, as it discovers that this function may invoke a `delete` function with `null` parameter. However, the performance bug is not related to this delete function call.

Table 2.3 Coverage comparison for all performance bugs and the matching frequency of each Hytrace static rule. "perf.": using only performance-related patterns/rules in Infer and Findbugs; "all": using all patterns/rules in Infer and Findbugs. "-": not supporting applications in specific languages (Caramel and Findbugs), or not implemented by Hytrace static (R3: unsafe function only checks C library functions).

System name	Total # bugs	Coverage						# of bugs matched by each rule				
		Hytrace-static	Infer (all)	Infer (perf.)	Findbugs (all)	Findbugs (perf.)	Caramel	R1	R2	R3	R4	R5
Apache	13	100%	0%	0%	-	-	0%	12	9	2	3	13
Lighttpd	7	100%	0%	0%	-	-	0%	7	2	0	2	6
Memcached	1	100%	0%	0%	-	-	0%	1	1	0	0	1
MySQL	19	100%	11%	5%	-	-	5%	18	18	2	7	17
Squid	13	100%	0%	0%	-	-	0%	13	6	0	1	13
Cassandra	27	100%	44%	44%	0%	37%	-	9	3	-	26	1
HDFS	18	100%	39%	39%	0%	17%	-	13	4	-	17	6
Mapreduce	28	100%	59%	59%	48%	57%	-	21	13	-	26	14
Tomcat	7	100%	43%	43%	14%	43%	-	6	2	-	3	1

Table 2.2 also shows the false positives of different schemes. The “TP” means whether each scheme has identified bug-related functions. The “FP" means the number of reported functions by each scheme, which are not related to the corresponding performance bug. For PerfScope, Hytrace-dynamic, and Hytrace which have the ranking mechanisms, the “FP” is the number of reported functions which are not related to the corresponding performance bugs and have higher rank than or the same rank as the bug-related functions. Overall, Hytrace produces the fewest false positives with the highest coverage among all techniques in comparison. It validates our hypothesis that combining static code pattern checking and runtime anomalous behavior detection achieve better bug diagnosis precision than pure static or pure dynamic techniques. Infer and Findbugs incur large false positives in anomaly diagnosis, especially for all tested Java bugs, mainly because their checking is not guided by specific performance anomaly. Since they are designed for general bug detection not anomaly diagnosis, they simply report all suspicious code regions, regardless whether these code regions are related to the performance anomalies under diagnosis or not. Note that, many of these false positives in Table 2.2 could be true bugs or bad programming practices. However, they are not related to the performance anomalies under diagnosis. Moreover, Infer encounters runtime execution errors in 4 bugs.

Even though Hytrace's result is much better than other static tools (i.e., Infer, Findbugs, Caramel) in Table 2.2, we do not mean that Hytrace can replace them. We know that those tools have different targets, but they are the best static performance-related tools that we can find.

To further evaluate the generality of Hytrace static rules, we applied the static component of Hytrace to all the 133 real-world performance bugs we could find on JIRA and Bugzilla. Note that, evaluating Hytrace-dynamic component requires us to reproduce the bugs. Since it is impractical to reproduce hundreds of real-world performance bugs given the complexity of the performance problems and time limitation, the evaluation presented below reflects our best effort of evaluating the generality of Hytrace static rules.

As shown in Table 2.3, Hytrace static rules provide 100% coverage for all 53 performance problems in C/C++ programs. That is, for each of these 53 performance problems, at least one of the five Hytrace static rules can identify a bug-related function as a suspicious function (i.e., *potential* root cause). We call a function f related to a bug b if developers modify f to fix b. In comparison, other tools have poor coverage for these C/C++ performance problems. Although Findbugs and Infer perform better for Java performance problems, the best coverage they can achieve is still below 60% (e.g., Mapreduce). In contrast, Hytrace-static also achieves 100% coverage for all 80 Java bugs. The average number of reported functions by each scheme for the 133 bugs in Table 2.3 is similar to the average number of false positive functions for the 14 bugs in Table 2.2. Hytrace-static reports more potential root cause functions than other static schemes, which matches our design principle—Hytrace static rules favor generality over precision to achieve high coverage.

Table 2.3 also shows the number of *potential* root cause that are covered by each Hytrace static rule. As we can see, “R1: constant parameter” rule and “R5: uncovered branch” rule cover the most C/C++ bugs, while “R4: unchanged loop exit condition variables” rule covers the most Java bugs.

Table 2.4 The rank of root cause functions identified by different schemes. Smaller numbers mean higher ranks.

Bug name	Hytrace						PerfScope Rank
	Rank	Matched rules					
		R1	R2	R3	R4	R5	
Apache-37680	7	✓	✓	✗	✗	✓	15
Apache-43238	7	✓	✓	✗	✗	✓	9
Apache-45856	1	✓	✗	✓	✗	✓	41
Lighttpd-1212	1	✓	✗	✗	✗	✗	1
Lighttpd-1999	2	✓	✗	✗	✗	✓	2
Memcached-106	2	✓	✓	✗	✗	✓	4
MySQL-54332	2	✓	✓	✗	✗	✗	3
MySQL-65615	2	✓	✗	✗	✗	✓	5
Cassandra-5064	2	✓	✗	✗	✓	✗	4
Mapreduce-3738	4	✓	✓	✗	✓	✗	12
HDFS-3318	2	✓	✗	✗	✓	✓	5
Tomcat-53450	1	✗	✗	✗	✓	✗	2
Tomcat-53173	10	✓	✗	✗	✗	✓	14
Tomcat-42753	2	✓	✓	✗	✗	✗	29
Avg.	3	93%	43%	7%	29%	57%	10

The “R3: unsafe function” rule covers the least bugs, and does not cover any Java bugs, as our current prototype only includes a few C library functions as unsafe.

The reasons we use *potential* root cause instead of *real* root cause in Table 2.3 are: 1) Hytrace-static only captures the *potential* buggy functions who are vulnerable to the performance problems; 2) Hytrace outputs the *real* root cause functions relying on both static and dynamic results; and 3) in order to use Hytrace-dynamic analysis on those 133 bugs, we have to reproduce them first, which is time-consuming (Section 4.3.1).

Table 2.4 provides a detailed comparison of Hytrace and PerfScope, showing the bug-related functions identified by them and their ranks. Smaller rank-number means higher rank: “1” means the highest rank. The results show that Hytrace can significantly improve the ranking of all the bug related functions with exceptions only when the bug related functions are already ranked the first or the buggy function is not detected (one false negative case). The benefit of the rank improvement is significant because the developer might spend lots of time on examining those false alarm functions that are ranked before the true root cause function. By increasing the rank of the root cause function, we can potentially cut down the performance diagnosis time a lot (e.g., the rank of root cause function in Apache-45856 is increased from the 41st to the 1st).

Hytrace achieves the rank improvement from two aspects: 1) filtering out many false positive functions identified by the purely dynamic scheme that do not exhibit any static bug characteristics, which boost the ranks of those true bug-related functions; and 2) some lower ranked bug related functions are actually the immediate caller of higher ranked functions. Because of the rank inheritance, by adding the immediate callers of identified bug related functions, those bug related functions get higher ranks.

Table 2.5 The rank of root cause functions and the number of false positive functions identified by different schemes. Smaller numbers mean higher ranks.

Bug name	PerfScope		Hytrace		Hytrace with							
					2-hop callers		3-hop callers		4-hop callers		5-hop callers	
	Rank	FP	Rank	FP	Rank	FP	Rank	FP	Rank	FP	Rank	FP
Apache-37680	15	14	7	17	7	34	7	51	1	39	1	51
Apache-43238	9	8	7	6	7	12	7	14	5	17	5	25
Apache-45856	41	40	1	5	1	107	1	181	1	205	1	208
Lighttpd-1212	1	0	1	2	1	11	1	14	1	27	1	34
Lighttpd-1999	2	1	2	4	2	8	2	21	2	34	2	42
Memcached-106	4	3	2	2	1	2	1	2	1	2	1	2
MySQL-54332	3	2	2	6	2	163	1	196	1	228	1	228
MySQL-65615	5	4	2	2	4	44	4	55	3	55	3	55
Cassandra-5064	4	3	2	1	3	5	3	15	1	21	1	29
Mapreduce-3738	12	11	4	7	4	13	4	15	4	16	4	17
HDFS-3318	5	4	2	2	3	6	3	27	1	5	1	12
Tomcat-53450	2	1	1	8	2	32	2	157	2	245	2	315
Tomcat-53173	14	13	10	15	10	28	5	100	5	207	4	276
Tomcat-42753	29	28	2	2	2	14	2	26	3	118	3	196
Avg.	10	9	3	6	4	34	3	62	2	87	2	105

2.5.2 Sensitivity Study

We conducted a sensitivity study in order to determine how multi-hop caller functions added in the Hytrace-dynamic list affect the coverage and precision of Hytrace diagnosis.

Our results in Table 2.5 show that adding more hops of caller functions has little improvement over the rank of the root cause functions but significantly increases the false positives.

2.5.3 Case Study

To further understand how the output of Hytrace can be used for debugging, we now discuss bug inference results in detail. We pick 5 representative cases to cover both C/C++ and Java applications.

Apache-37680 (C/C++): The patch and the cause of this bug (Figure 2.1) is already discussed in Section 2.1.1. As mentioned earlier, a graceful restart after some configuration change hangs Apache server. The direct cause of the hang is that function `child_main` is stuck in a re-try loop. This loop keeps issuing a blocking call `accept` to a socket until the blocking call succeeds. Unfortunately, since the target socket is configured to not allow blocking calls, `accept` always returns `EWOULDBLOCK/EAGAIN`, and the loop never exits. The root cause of this hang is that the graceful restart did not change the configuration of the reused socket from not allowing blocking calls to allowing blocking calls. This root cause is inside function `ap_setup_listeners` shown in Figure 2.1. The buggy code in `apr_socket_opt_set` only allows non-blocking calls through the constant parameter '1'.

Hytrace effectively identified all the three key functions related to this performance problem, `child_main`, `apr_socket_opt_set`, and `ap_setup_listeners`, and ranked them the 7th, 14th, and 14th respectively. In comparison, PerfScope only identified `apr_socket_opt_set` and ranked it the 15th. PerfScope did not identify either `ap_setup_listeners` or `child_main`, because both of them do not produce many system calls. Hytrace-dynamic can identify those two root cause related functions by adding the caller function of `apr_socket_opt_set`, which is `ap_setup_listeners`, and the caller function of `proc_mutex_sysv_acquire`, which is `child_main`. Finally, Hytrace rule checking results show that `child_main`, `ap_setup_listeners`, and `apr_socket_opt_set` all match one or multiple performance-problem-prone rules. They are kept in the suspicious function list. After removing originally higher ranked functions by matching Hytrace static rules, the ranks of these three root cause related functions all rise to top 14.

In this case, Hytrace report exactly reflects the root cause. As discussed in Section 2.1.1, the patch exactly changes the "constant parameter", 1, passed to the invocation of `apr_socket_opt_set` in function `ap_setup_listeners`.

Apache-45856 (C/C++): In this bug, when `suexec_log` is larger than 2GB, corresponding CGI and SSI applications, which are using the suEXEC feature of Apache server, hang. The root cause of this hang is in function `err_output`, which uses `fopen` to open and append large files (larger than 2GB) on a 32-bit machine.

Hytrace can identify the root cause function `err_output` and have improved this function's

```
//ColumnFamilyStore.java                                    Cassandra-5064(v1.2.0)
143 public void reload() { /*bug related function*/
      ...
168   while (true) {
        ...
172     if (memtable.initialComparator == comparator)
173       break;
174
175     Future future = maybeSwitchMemtable(getMemtableThreadSafe(), true);
        ...
178   }
      ...
188 }

614 public Future<?> maybeSwitchMemtable(Memtable oldMemtable, final boolean
    writeCommitLog) {  /*bug related function*/
      ...
649   Memtable mt = cfs.getMemtableThreadSafe();
650   if (!mt.isClean() && !mt.isFrozen()) {
        ...
        icc.add(cfs);
654   }
      ...
658   for (ColumnFamilyStore cfs : icc) {
        ...
662     memtable.flushAndSignal(latch, flushWriter, ctx); //flush memtable list
663   }
      ...
690   if (writeCommitLog) {
        ...
694     ...discardCompletedSegments(metadata.cfId, ctx.get()); //CPU intensive
695   }
      ...
703 }
```

Figure 2.7 Partical call graph for the Cassandra-5064 bug. "⟶" represents the function call invocation.

ranking significantly from the PerfScope result (41th up to 1st). Specifically, when the performance anomaly happens, Hytrace-dynamic identifies both function `err_output` and its caller `log_err`, which invoke system calls with abnormal frequencies. Hytrace rule checker has kept the root-cause function `err_output` in its result, because this function matches "constant parameter", "unsafe function", and "uncovered branch" rules. With some functions, which were originally higher ranked by Hytrace-dynamic, not matching any Hytrace rules, the ranks of `err_output` and its caller gets improved a lot. Clearly, the "unsafe function" rule matched with `err_output` is exactly the root cause.

Cassandra-5064 (Java): Users reported that sometimes Cassandra would hang as soon as an `ALTER TABLE` request is issued. The hang actually happens in a `while` loop in `reload` function, as shown in Figure 2.7. In this loop, `maybeSwitchMemtable` processes every memtable in a list (line 174), until there is no remaining memtable in the list (line 172–173). Clearly, `maybeSwitchMemtable` should remove a memtable `mt` from the list after `mt` is processed. Unfortunately, this is only done for dirty memtables (line 650–652), but not clean memtables. As a result, the `while` loop in `reload` becomes infinite, where `maybeSwitchMemtable` keeps getting invoked to process the same clean memtable again and again, endlessly.

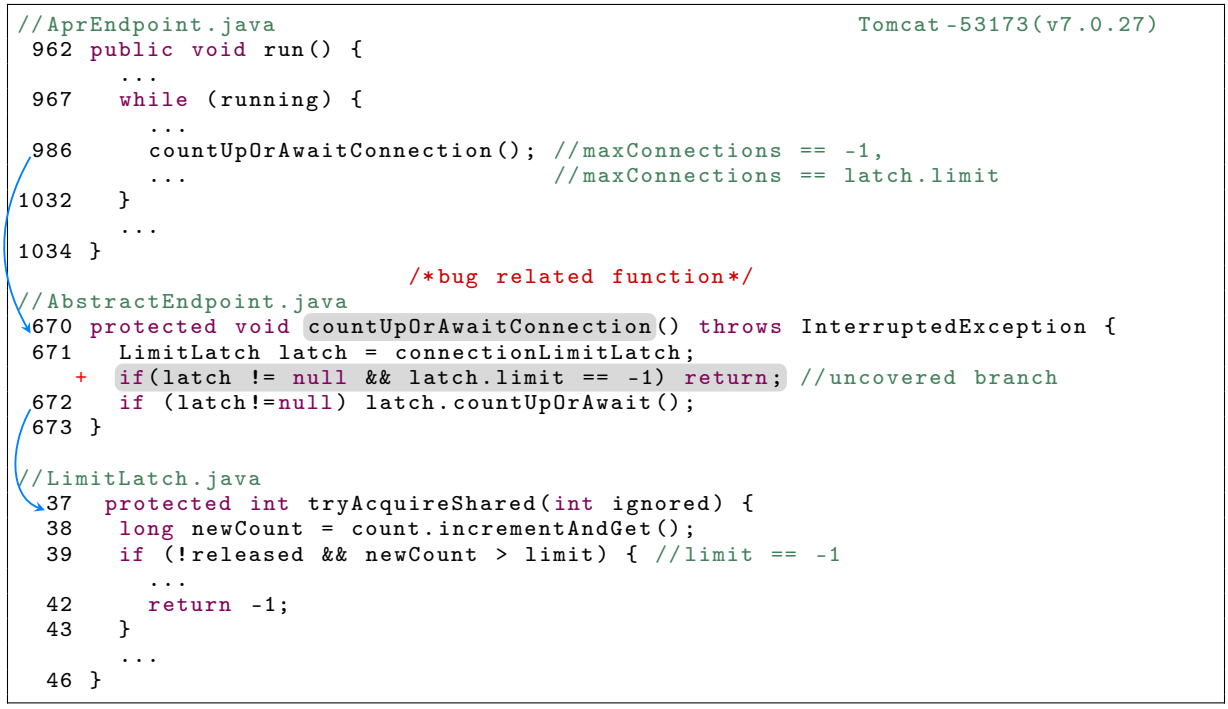

Figure 2.8 Partial call graph for the Tomcat-53173 bug. "⟶" represents the function call invocation. "+" means added lines, representing the patch for the bug.

Hytrace identified both `maybeSwitchMemtable` and `reload` as rank two suspicious functions. Specifically, Hytrace-dynamic detected `maybeSwitchMemtable` because certain system calls are invoked much more frequently when the bug is triggered. Hytrace-dynamic then adds `reload` to the suspicious function list, because it is the caller of `maybeSwitchMemtable`. Hytrace rule checker did not prune out these two functions, as they both match the "constant parameter" rule, and `reload` also matches the "unchanged loop exit condition variables" rule.

The "constant parameter" rule matched with `reload` is the direct cause, while the "unchanged loop exit condition variables" rule matched with `reload` is related to the root cause of the observed performance problem. Specifically, `reload` invokes `maybeSwitchMemtable` with a constant parameter, `True` (line 174). As a result of this constant `True`, expensive `CommitLog.discardCompletedSegments` function is always invoked inside `maybeSwitchMemtable` (line 694). And all of the above operations keep happening in the `while` loop (line 168) without updating any loop exit condition variables, consuming a lot of CPU and disk resources and causing the performance problem observed by users.

Tomcat-53173 (Java): Users reported that sometimes Tomcat would hang as soon as `maxConnections` is set to be `-1`. The hang happens because Tomcat is stuck inside the `countUpOrAwaitConnection` function, as shown in Figure 2.8 (the value of `maxConnections` is passed to `latch.limit` and `limit`). When `Acceptor` thread processes incoming connections, it calls function `countUpOrAwaitConnection` (line 986). In theory, setting `maxConnections` as `-1` means putting no upper-limit to accepting client socket connections. Consequently, `countUpOrAwaitConnection` should

```
//AppLogAggregatorImpl.java                                    Mapreduce-3738(v0.23.1)
137   public void run() { /*bug related function*/
   +    try {
          ... //runtime exception
   +    } finally {
193       this.appAggregationFinished.set(true);
   +    }
194   }

249   public void join() { /*bug related function*/
        ...
253     while (!this.appAggregationFinished.get()) {
254       LOG.info("Waiting for aggregation to complete");
255       try {
256         Thread.sleep(THREAD_SLEEP_TIME);
257       } catch (InterruptedException e) {
            ...
261       }
262     }
263   }
```

Figure 2.9 Partial call graph for the Mapreduce-3738 bug. "+" means added lines, representing the patch for the bug.

return immediately without any waiting. Unfortunately, this special setting (i.e., `-1`) is not specially handled. Instead, function `latch.countUpOrAwait` is invoked to try fetching a lock. This lock fetching will never succeed, as indicated by line 39 and line 42 in function `tryAcquireShared` — when `limit` is `-1`, the line-39 condition is always true and hence the function always returns `-1`, indicating a lock-acquisition failure. The execution then gets stuck in repeatedly trying to acquire the lock, while the client's connections get blocked.

Hytrace identified `countUpOrAwaitConnection` as a rank 10 suspicious function. Specifically, Hytrace-dynamic detected `countUpOrAwaitConnection` because it invokes a set of system calls with abnormal frequencies in performance-anomaly runs. as it matches with the "uncovered branch" rule — line 672 in Figure 2.8.

The uncovered-branch rule matched with `countUpOrAwaitConnection` is related to the root cause of the observed performance problem. The patch exactly added more handling for more branch scenarios around line 672, as shown in Figure 2.8.

Mapreduce-3738 (Java): In our previous paper [Dai17], Hytrace failed to diagnose the Mapreduce-3738 bug. we have re-done the experiments and found that the miss detection is caused by missing the profiles for the root cause functions. After adding the missing profiles back, Hytrace successfully identifies the root cause function `AppLogAggregatorImpl.join` and ranked it the 4th.

We now describe this bug in details. As shown by Figure 2.9, once an uncaught runtime exception (e.g., `OutofMemoryError`) happens in the function `AppLogAggregatorImpl.run`, the true-setting for a variable `appAggregationFinished` could be skipped (line 193). NodeManager will then hang during shutdown by calling `AppLogAggregatorImpl.join`, waiting for `appAggregationFinished` to become true forever (line 253). The patch simply moves the `set(true)` into a `finally` block, which guarantees the execution of `set(true)` even when an uncaught exception happens.

Table 2.6 Performance of Hytrace-static program analysis and Hytrace-dynamic trace analysis (the run-time workload is described in Section 4.3.1).

Bug name	Static analysis time (sec)	Application lines of code (K)	Dynamic analysis time (min)	Trace size (MB)
Apache-37680	5.9 ± 0.02	266.7	1.3 ± 0.01	406
Apache-43238	4.5 ± 0.01	312.8	3.5 ± 0.01	306
Apache-45856	4.8 ± 0.02	314.7	2.4 ± 0.01	324
Lighttpd-1212	2.1 ± 0.01	53.9	1.1 ± 0.01	337
Lighttpd-1999	3.0 ± 0.01	58.4	13.1 ± 0.02	1,365
Memcached-106	29.2 ± 0.01	11.0	25.6 ± 0.32	3,603
MySQL-54332	9.6 ± 0.01	1,233	9.2 ± 0.41	316
MySQL-65615	12.9 ± 0.03	1,759	5.8 ± 0.12	77
Cassandra-5064	29.2 ± 2.23	259.0	21.7 ± 0.40	1,054
Mapreduce-3738	40.5 ± 3.02	935.8	16.0 ± 0.20	550
HDFS-3318	108 ± 0.60	1,114	15.7 ± 1.22	473
Tomcat-53450	35.2 ± 0.38	407.9	5.7 ± 0.34	35
Tomcat-53173	49.7 ± 0.40	405.0	2.0 ± 0.02	143
Tomcat-42753	49.5 ± 0.20	456.9	9.1 ± 0.80	274
Avg.	27.4 ± 0.50	542.0	9.4 ± 0.28	662

Hytrace identifies `AppLogAggregatorImpl.join` as a suspicious function and have improved this function's rank from the PerfScope result (12th to 4th). Specifically, Hytrace-dynamic detected `AppLogAggregatorImpl.join` because it invokes system call sequence {`sys_futex`, `sys_stat64`, `sys_stat64`, `sys_futex`} with abnormal frequencies. Hytrace rule checker did not prune out this function, as it matches with the "unchanged loop exit condition variables" rule (line 253). In addition, Hytrace rule checker also identifies `AppLogAggregatorImpl.run` as a suspect function, as the invocation of `set(true)` matches the "constant parameter" rule. However, the dynamic component of Hytrace fails to identify `run` function. The reason is that Hytrace-dynamic looks for abnormal system-call related runtime behavior changes. `AppLogAggregatorImpl.run` itself and its callee functions do not issue many system calls and hence are not identified as abnormal.

2.5.4 Hytrace Overhead

Hytrace-static is efficient in its program analysis, benefiting from the simplicity of its rules. It takes less than a minute to process most applications in our experiments. For the largest software in our experiments, HDFS with more than 1 million lines of code, Hytrace finishes the static analysis in about 100 seconds. Note that, Hytrace-static only needs to process each program once for all the performance anomaly diagnosis one might want to do inside the program.

Hytrace-dynamic needs to collect system-level metrics through LTTng at run time and then analyze the corresponding trace. Its run-time CPU overhead is always less than 3%. Its trace analysis time depends on the trace size. As shown in Table 2.6, it can finish analyzing hundreds of mega-bytes of traces usually within a couple of minutes. The core part of the Hytrace trace analysis, frequent

episode mining, can be easily parallelized and achieve much better performance, if needed. Note that, our evaluation uses workloads described in Section 4.3.1. To trigger the performance problems under diagnosis, we could have used shorter-running workloads, which would take less analysis time for Hytrace-dynamic.

2.6 Limitation Discussion

Our current evaluation focuses on single node performance bugs. For distributed performance bugs, Hytrace's diagnosis schemes are still preliminary. It generates a consolidated buggy function list by taking the intersection among all the buggy function lists produced by different faulty nodes. However, distributed system bugs can manifest as a chain of abnormal functions over multiple dependent nodes. Hytrace currently does not consider such causal relationships between distributed components. Previous work (e.g., FChain [Ngu13a], PCatch [Li18]) has developed distributed bug diagnosis tools based on distributed system causal analysis. Hytrace can integrate with those tools to achieve more precise distributed system performance bug diagnosis.

Hytrace-static component currently has five generic rules. Although our rule set can achieve 100% coverage on the 133 performance bugs, we do not claim that those five rules can identify all the performance problems reported by production cloud users. Hytrace framework allows users to easily add new rules with few code changes. Furthermore, we currently did not find any unsafe function in Java programs which matches our rule R2. We plan to extend this rule by adding more I/O related functions in Java, which is part of our future work.

Hytrace-dynamic integrates runtime execution path analysis with abnormal function detection to achieve high coverage. However, it cannot identify all the root cause functions in every case. For example, in Mapreduce-3738 bug (Section 2.5.3), Hytrace-dynamic identifies the `join` function but fails to identify the `run` function because the `run` function and its callee functions produce few system calls during runtime. The miss detection can be addressed by integrating data flow analysis into Hytrace-dynamic. For example, we can add `run` function into the candidate function list because both `join` and `run` perform operations on the same data (i.e, `appAggregationFinished`), which is also part of our future work.

2.7 Summary

In this chapter, we have presented Hytrace, a hybrid approach to diagnosing real-world performance bugs in production cloud systems. Hytrace combines rule based static analysis and runtime inference techniques to achieve higher accuracy than pure-static or pure-dynamic approaches. Hytrace does not require any application source code or instrumentation, which makes it practical for production cloud environments. We have implemented a prototype of Hytrace and tested it over 133 real performance bugs discovered in different commonly used server applications. Our results show that Hytrace can greatly improve coverage and precision comparing with existing state-of-the-art

techniques. Hytrace is light-weight, which imposes less than 3% CPU overhead to the testing cloud environments.

CHAPTER

3

DSCOPE: DETECTING REAL-WORLD DATA CORRUPTION HANG BUGS IN CLOUD SERVER SYSTEMS

3.1 Introduction

Cloud server systems such as Hadoop and Cassandra [Had; Cas] have enabled many real-world data-intensive applications ranging from security attack detection to business intelligence. However, due to their inherent complexity, those cloud server systems present many performance challenges. Particularly, previous studies [Gun14; Gan17] have shown that many tricky performance bugs in cloud server systems are caused by unexpected data corruptions which are more likely to be overlooked by the developer. For example, in May 2017, a data corruption bug triggered in a data center failover operation brought down the British Airway service for hours [Bac].

Performance bugs[1] are notoriously difficult to debug because they typically produce little useful debugging information. The problem exacerbates in cloud server systems since the developer typically does not have the access to the original input data that triggered the performance bug or the large scale infrastructure to replay the failed production run. Although previous work has extensively studied data corruptions (e.g., [Gan17; Bai08a; Bor11; Jia08; Wan15]) and performance bugs (e.g., [Hua15; Kal17; Son17; Dai18b]), little research has been done to study the intersection

[1]We use performance bugs to broadly refer to all non-functional bugs, which could cause slowdown or system unavailability.

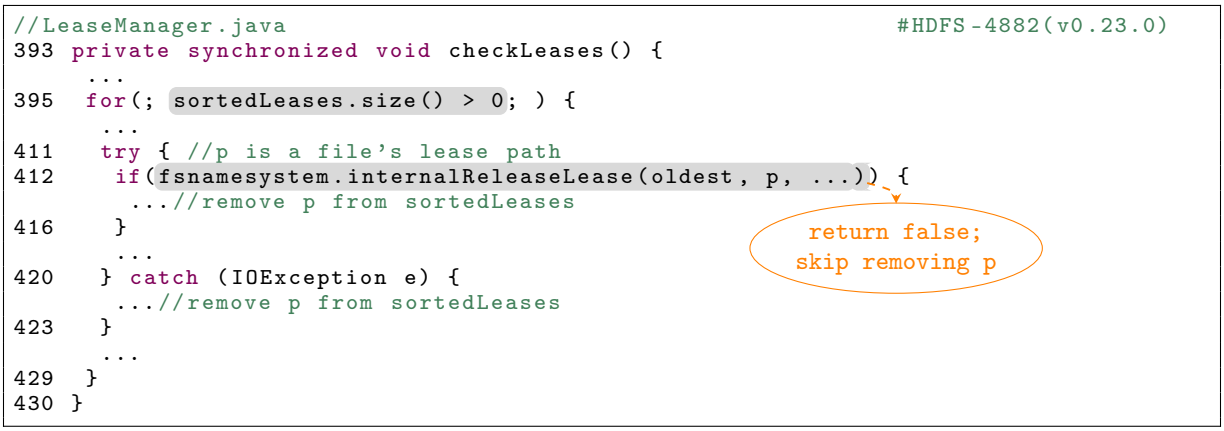

Figure 3.1 A real-world data corruption hang bug from HDFS. A corrupted file `f` associated with the lease path `p` makes the `internalReleaseLease` function fail for recovering the lease for `f`. When this failure happens, `p` is not removed from `sortedLeases` (skip updating loop index), LeaseManager keeps recovering lease for the file f endlessly. "- ->" represents the control flow.

between the two, that is, the performance problems caused by data corruptions. Particularly, our work focuses on detecting software hang bugs that are triggered by data corruptions in cloud server systems. Software hang bugs make the system become unavailable to either part of or all of the users, which is one of the most severe performance problems production systems try to avoid [Dea15a; Jin12b; Dea14; Dai18b].

3.1.1 A Motivating Example

To better understand how real-world data corruption hang bugs happen, we use a known HDFS-4882 bug as one example shown by Figure 4.13. This hang bug happens when the improper handling of a corrupted file `f` causes the loop to skip updating its loop index. In HDFS, when a client's leases get expired, the lease recovery is triggered by the LeaseManager on the NameNode. The LeaseManager sends a lease recovery request for each lease in the `sortedLeases` set to the `FSNamesystem` via a RPC call (line #412-413). If a lease is successfully recovered (e.g., released or renewed) (line #412-416), or an `IOException` happens during the lease recovery (line #420-423), the lease path `p` is removed from the `sortedLeases`. The LeaseManager keeps recovering and removing the leases until the `sortedLeases` set is empty (line #395). However, the `FSNamesystem` only considers the case where the last block is corrupted if data corruption happens in a file. Thus, when the second-to-last block of a file `f` (i.e., an `INode`) is corrupted but the processing state of the last block of `f` is complete, the `FSNamesystem` improperly handles this case and returns `false` by mistake (line #412). This bug occurs when the HDFS client finished writing the second-to-last block, starts to write the last block and part of the DataNodes experience a shut-down failure. To resume the process, the NameNode marks the second-to-last block as committed, unblocks the HDFS client from writing the last block, and marks the last block as complete [Hdf]. As a result, the lease path `p` is not removed from the `sortedLeases`, and the LeaseManager keeps invoking the lease recovery for the same

lease endlessly.

3.1.2 Our Contribution

This chapter presents DScope, an automated corruption-hang bug detection tool for server systems commonly used in computing clouds. DScope is a static analysis tool — it can detect data-corruption related hang bugs without running the target system and it requires no system-specific knowledge. To achieve both high coverage and low false positives, DScope first uses static control flow and data flow analysis to identify loops whose exit conditions may be affected by external data (i.e., I/O operations), and then conducts loop bound and loop stride analysis to filter out loops which are guaranteed not to have hang problems. To support such analysis, DScope models Java data-related APIs which are commonly used in cloud server systems.

This chapter makes the following contributions:

- We present a hang-bug detection scheme that identifies potential infinite loops caused by data corruptions in cloud server systems.
- We describe a false-positive pruning technique that identifies always-exit loops through loop stride and bound analysis. Different from generic loop analysis, our analysis focuses on a wide variety of Java I/O APIs widely used in cloud systems, and helps greatly improve the accuracy of our data-corruption hang-bug detection.
- We categorize real-world data-corruption hang bugs into four common types based on DScope detection results. This categorization will help future work on avoiding, detecting, and preventing data-corruption hang bugs.

We have implemented DScope and evaluated it using 9 commonly used cloud server systems (e.g., Cassandra, HDFS, Mapreduce, Hive, etc). DScope reports 42 true data corruption hang bugs, with 29 of them are newly discovered bugs. We also applied two state of the art static bug detectors, Findbugs [Fin] and Infer [Fbi], to the same set of systems. They detect very few corruption hang bugs (2 for Findbugs and 1 for Infer), indicating the need for a dedicated corruption hang bug detector like DScope.

3.2 System Design

This section first provides an overview of DScope (§3.2.1). It then presents the detailed designs of how to discover corruption-hang bug candidates (§3.2.2) and how to prune false positives (§3.2.3).

3.2.1 Approach Overview

DScope focuses on detecting software hang bugs caused by potential data corruptions in cloud server systems. Our bug detection scheme consists of two major steps: 1) discovering all candidate data corruption hang bugs that aims at maximizing detection coverage; and 2) filtering out false

```
549 for (int j = 0; j < length; j++) {
550   String rack = oneblock.racks[j];
        ...
559 }
560
```

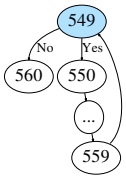

Figure 3.2 The example of a simple loop with the source code block and the corresponding CFG in the `CombineFileInputFormat` class in Hadoop v0.23.0.

positive detections by identifying code patterns that assure the program will not hang under any circumstance.

Since many software hang problems are caused by infinite loop bugs [Dea15a; Jin12b], our work focuses on detecting possible infinite loops caused by data corruptions in cloud server systems written in Java. To detect those loop bugs, DScope leverages the Soot compiler framework [Soo] to compile application bytecode into intermediate representation (IR) code (i.e, Soot Jimple) and perform static analysis over the IR code in three steps: 1) loop path extraction, 2) I/O depdent loop identification, and 3) loop stride and bound analysis.

Specifically, DScope first extracts different execution paths which start from the loop header and end at the loop header by traversing the control flow graph (CFG) of all loops. Next, DScope derives the exit conditions of each loop path and checks whether those exit conditions depend on any I/O operations. The rationale is that if the loop exit condition depends on an I/O operation, a data corruption (e.g., hardware failure [Bai08a; Hwa12; Sch09; Bai08b; Ole16; Kad09], software fault [Wan15]) can cause the loop exit condition to be never met and thus an infinite loop software hang bug.

After discovering candidate data corruption hang bugs, DScope performs false positive pattern filtering to improve the bug detection precision. The false positive filtering is based on the loop index, loop stride (i.e., the delta value applied to the loop index in each iteration) and loop bound analysis on every loop path. For example, if the loop stride is always positive when the loop bound is an upper bound (or if the loop stride is always negative when the loop bound is a lower bound), and if the loop bound is unchanged during each loop iteration and the loop exit conditions involve bound checking, we say that the detected hang bug is a false positive because the loop will always exit without causing a software hang.

3.2.2 Identify Bug Candidates

DScope discovers candidate corruption-hang bugs by 1) traversing the CFGs of all loops in application functions to derive their loop paths and 2) checking whether the exit conditions of those loop paths are I/O dependent.

Loop path extraction. For a simple loop, the execution path within one loop iteration, called a *loop path*, consists of all the statements that start from the loop header and end at the loop header. We can extract the loop path easily by traversing the CFG of the loop. For example, Figure 3.2 shows

```
544 for(OneBlockInfo oneblock : blks){
      ...
549   for (int j = 0; j < length; j++){
550    String rack = oneblock.racks[j];
       ...
559   }
560
      ...
571 }
```

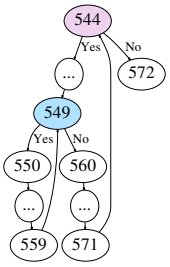

Figure 3.3 The example of nested loops with the source code block and the corresponding CFG in the `CombineFileInputFormat` class in Hadoop v0.23.0.

the source code and its CFG of a simple loop[2]. DScope generates a loop path $\{549, 550, ..., 559, 549\}$.

For a nested loop, the loop path consists of concatenations of the execution paths of both inner loops and outer loops. DScope extracts all loop paths using three steps. First, for the execution path, denoted as P_{outer} whose tail is the outer loop header, we add the path into a path set called S_{path}. Second, for the execution path P_{outer} whose tail is a loop body statement, we infer this statement must be the header of an inner loop and extracts the inner loop execution path denoted as P_{inner} from P_{outer}. Third, for each loop path in S_{path}, DScope first clones it and replaces any statement s_i with P_{inner} if s_i is an inner loop header and the current loop path does not contain P_{inner}. This new concatenated loop path is then added to the path set S_{path}. DScope repeats the third step until there is no more new loop path generated. Figure 3.3 shows an example of nested loops. First, DScope extracts one loop path $\{544, ..., 549, 560, ..., 571, 544\}$. Second, DScope extracts $\{549, 550, .., 559, 549\}$ as an inner loop path. Third, DScope replaces 549 with $\{549, 550, ..., 559, 549\}$ on $\{544, ..., 549, 560, .., 571, 544\}$, to create a new loop path $\{544, ..., 549, 550, ..., 559, 549, 560, ..., 571, 544\}$.

The third group of complicated loops involve exceptions. For those loops, some sub-paths become infeasible due to the exception handling, which should not be considered in our loop exit condition checking. For example, Figure 3.4 shows a `while` loop containing exception handling. The assignment statement at line #130 can throw an exception when the operation on the right hand side processes a `null` argument. As a result, the variable `key` is not updated and remains to be the default value which is `null`. So when the exception is triggered, the `if` statement (line #186) always returns `true`. Thus, all the statements in the else branch (line #188-206) are unreachable and any path consists of those statements are infeasible paths. In this example, DScope only generates the loop path as $\{120, 128, 129, 130, 139, 140, 141, 185, 186, 187, 207, 255, 256, 120\}$.

It can be computationally expensive to traverse the CFG of the loops containing multiple exceptions because every statement in the `try` block has two branches (i.e., triggering or not triggering the exception) resulting in a large CFG. DScope addresses the problem by grouping all the statements based on the data they process. Specifically, DScope identifies all the statements which involve function invocations in the `try` blocks and groups them based on the *arguments*

[2]DScope analyzes IR code directly to extract the execution path of different loops. For easy understanding, we illustrate the execution paths using source code in the rest of the chapter.

```
120 while (!dataFile.isEOF()){
      ...
128   DecoratedKey key = null;
129   try{
130    key =...;//throws Exception
      ...
139   } catch (Throwable th){
140    ...
141   }
      ...
185   try{
186    if (key == null)
187     throw new IOError(...);
188    if (dataSize > length)
       ...
206    outputHandler.warn(...);
207   } catch (Throwable th) {
       ...
255   }
256 }
257
```

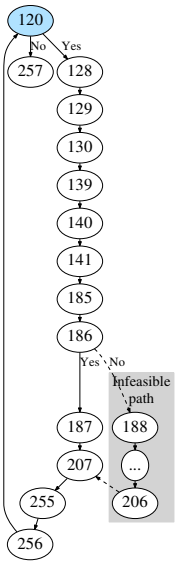

Figure 3.4 The example of a loop containing exception handling constructs with the source code block and the corresponding CFG in the `Scrubber` class in Cassandra v2.0.8.

```
//Soot IR
198 $i1 = r0.<InputStream: read()>(r2) //$i1 is an I/O related variable
199 if $i1 == -1 goto line #203 //$i1 == -1 is the exit condition
    ...
202 goto line #198
```

Figure 3.5 The example of the loop's exit condition directly depends on I/O operations. It is in the `IOUtils` class of Compress v1.0.

of those function invocations. Since DScope aims at detecting data corruption hang bugs, we can assume all the statements in the same group throw exceptions when their *arguments* get corrupted. Suppose there are m statements in the `try` blocks. DScope divides all m statements into n groups and runs the loop path discovery algorithm 2^n times. Thus, DScope can reduce the loop path search space from 2^m to 2^n ($n \ll m$), which reduces DScope's analysis time and resource requirements (e.g., avoiding analysis failures caused by `OutofMemoryException`).

I/O dependent loop identification. To discover candidate data corruption hang bugs, DScope identifies those loops whose exit conditions depend on I/O operations, which are called *I/O dependent loops*. After extracting a loop path, DScope identifies all the loop exit instructions and derives the loop path's exit conditions by performing a union over the exit conditions of all the branch statements. We consider a loop path is I/O dependent if *any* of its exit conditions depend on I/O operations. The rationale is that a data corruption can cause the corresponding I/O operations to return unexpected values or throw exceptions, making the loop never exit and thus software hang. DScope considers the operations performed on *I/O classes* via virtual invocations or on the *I/O variables* via instance invocations as *I/O operations*. The I/O classes include all classes and interfaces in `java.io` and `java.nio` packages and their subclasses and implementation classes. The instances of the I/O classes are called *I/O variables*. Additionally, DScope allows users to easily

```
//Soot IR
 3   if l8 >= l0 goto line #12 //l8 >= l0 is the exit condition
     ...
 5   $l2 = l0 - l8
 6   $l4 = $r2.<InputStream: skip>($l2)//$l4 is an I/O related variable
 7   $b5 = $l4 cmp 0L
 8   if $b5 == 0 goto line #12 //$b5 == 0 is the exit condition
 9   $l7 = $l8 + $l4
10   i8 = $l7
11   goto line #3
```

Figure 3.6 The example of the loop's exit condition indirectly depends on I/O operations. It is in the `NonSyncDataInputBuffer` class of Hive v2.3.2.

add application I/O classes in the configuration files to maximize detection coverage by identifying more application I/O dependent loops.

DScope checks whether the loop exit conditions *directly* depend on I/O operations by identifying the appearance of I/O classes in the exit checking statements. Figure 3.5 shows an example where the loop exit condition directly depends on the I/O operations. In this example, the variable `$i1` in the exit condition checking statement (line #199) is directly derived from a Java I/O class called `InputStream`.

DScope checks whether the loop exit conditions *indirectly* depend on I/O operations by performing data dependency analysis on all the statements of the corresponding application function. Specifically, DScope first identifies all the I/O related variables which are assigned with the return values of the I/O operations. Second, for each assignment statement of the application function that involves any I/O related variables on its right-hand-side, DScope iteratively labels the variable on the left-hand-side of the assignment statement as I/O related variables as well. After identifying all the I/O related variables, DScope checks whether the loop exit conditions are I/O dependent by identifying the appearance of I/O related variables in the exit checking statements. Figure 3.6 shows an example of indirectly I/O dependent loop exit condition. In this example, the loop exit checking involves `$l8` (line #3) and `b5` (line #8) whose value is derived from `l4` which is derived from a Java I/O operation `InputStream.skip()`.

To further check whether the loop exit conditions depend on I/O operations conducted on *complex I/O related variables* (i.e., variables with composite types), DScope performs an integrated analysis by linking variable information from IR code, Java source code, and Java bytecode [3]. DScope considers a variable with composite type as I/O related if any of the variable's elements is I/O related. Note that, by checking only the IR code, DScope might miss identifying some complex variables as I/O related. For example, in Figure 3.7, by checking only the IR code, DScope cannot identify the variable `$r13` as an I/O related variable, thus all the operations conducted on `$r13` will not be considered as I/O operations. This is because `$r13` is of type `HashMap` and `HashMap` is not an I/O class. To identify complex I/O related variables, DScope needs to retrieve the full type information (i.e., class path) in the Java bytecode for a target variable in the IR code.

[3] DScope mainly works on Soot IR code, except the integrated analysis in the I/O dependent loop identification module.

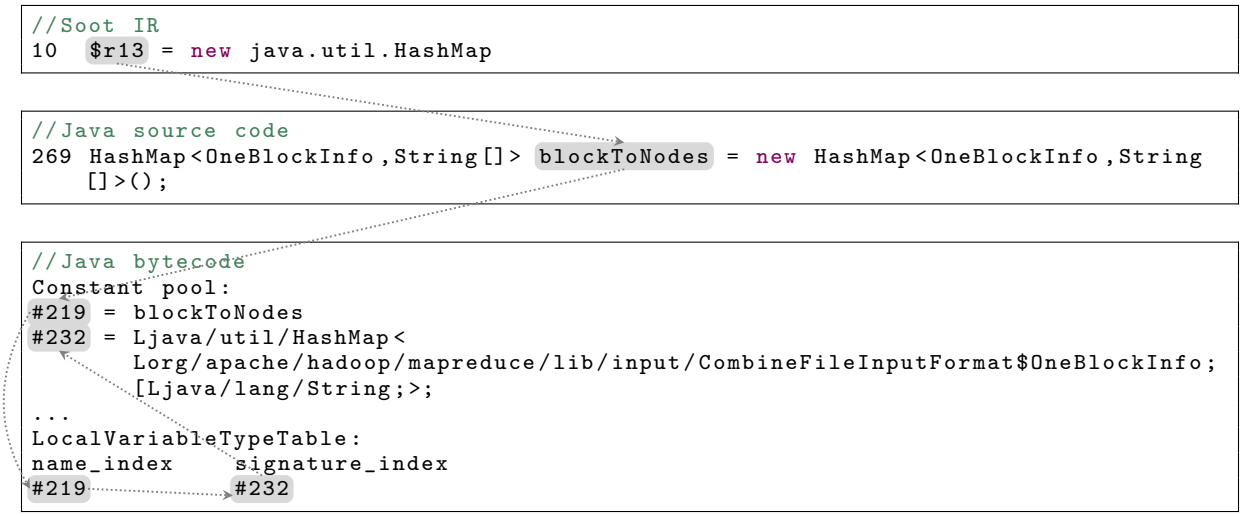

Figure 3.7 The `java.util.HashMap<K,V>` example in the `CombineFileInputFormat` class in Hadoop v0.23.0.

However, there is no direct mapping from IR code to Java bytecode. So, DScope has to leverage the source code to establish the mapping from. Specifically, DScope first retrieves the source code line number from Soot via `getLineNumber()` API for each variable val_{IR} in the IR code. DScope then analyzes the corresponding source code and extracts val_{IR}'s name in the source code, denoted as val_{src}. In Figure 3.7, DScope extracts that the variable `$r13` is defined at line #269 in the source code with name `blockToNodes`. Next, DScope leverages Coffi [Ver96], a Java bytecode parser, to extract the full type information for the target variable val_{src}. Specifically, the constant pool provides index lookup for each variable and LocalVariableTypeTable provides the mapping from the variable's index to the index of its signature which contains the full type information. In Figure 3.7, the constant pool indicates that the index of the variable `blockToNodes` is #219 and the LocalVariableTypeTable lookup tells its corresponding signature index is #232. DScope checks the constant pool using the index number #232 to derive the full type information of `blockToNodes`. Since `blockToNodes` consists of `CombineFileInputFormat$OneBlockInfo` class which is I/O related (i.e, an application I/O class), DScope infers `blockToNodes` is also I/O related. Thus the operations conducted on `blockToNodes` are I/O operations and the loops whose exit conditions depend on those I/O operations are I/O dependent loops.

3.2.3 Prune False Positives

Since our goal of candidate bug detection is to maximize coverage, false positives can be inevitably included in the candidate list. To improve DScope's bug detection precision, we further develop false positive pattern filtering schemes by identifying those loops which will always exit without causing any software hang. Our false positive filtering is achieved by analyzing the loop stride and loop bounds. DScope prunes false positive candidates by checking whether 1) the loop stride is *always* positive when the loop has an upper bound or the loop stride is always negative when the

Table 3.1 The 60 commonly used Java classes and interfaces which contain APIs related to the loop index, stride and bound.

Prefix	Class	# of classes or interfaces
java.io	DataInput family	2
	File	1
	InputStream family	12
	Reader family	10
java.nio	Buffer family	8
	channels.Channel family	20
java.util	Iterator, Enumeration	2
	List, Queue, Set, Stack	4
	StringTokenizer	1

loop has a lower bound; 2) the loop bound value is unchanged in *every* loop iteration; and 3) the loop exit conditions contain bound checking. Intuitively, any loops satisfying all those conditions will always exit without causing software hang, which should be pruned from DScope's detection list.

DScope's loop stride and bound analysis schemes consider two cases: a) the loop index, stride, and bounds are denoted by numeric primitives (e.g., integer); and b) the loop index, stride, and bounds are denoted by APIs in 60 commonly used Java classes and interfaces, shown in Table 3.1. Note that those Java classes and interfaces are not necessarily the *I/O classes* but appear frequently in the *I/O dependent loops*. Moreover, they do not include all the Java classes and interfaces which contain the loop related APIs. We plan to further extend our analysis to cover other Java classes in our future work, which can further improve our false positive filtering efficacy.

When the loop index, stride, and bounds are denoted by numeric primitives, DScope first extracts the loop index variable from the loop exit conditions. The *loop index* is a variable that appears in the loop exit conditions and is updated by another variable in an assignment statement via arithmetic operations (e.g., addition and subtraction). After identifying the loop index, the variable to which it compares in the exit conditions is the *loop bound* and the variable which is added to or subtracted from the loop index is the *loop stride*. DScope further checks whether the numeric stride is positive when the loop has an upper bound or negative when the loop has a lower bound. For example, in the expression "$index = index\ op\ stride$", if the "$op$" is an addition operation, the loop stride is positive. In the expression "$index\ symbol\ bound$", if the "$symbol$" is $\leq$ or $<$, the loop has an upper bound. Finally, DScope examines all the loop paths and checks whether the bound is unchanged within every loop path. Based on all the extracted information, DScope can make decision whether a discovered loop bug is a false positive.

When the loop index, stride, and bounds are denoted by *multiple* numeric primitives or the numeric primitives *outside* the current application function where the loop resides, DScope performs intra-procedure data flow analysis on all the statements of the corresponding application class to achieve accurate false positive filtering. For example, Figure 3.8 shows a loop with multiple strides

```
//Soot IR
127 $b6 = i1 cmp i0   //i1 is the loop index; i0 is the upper bound
128 if $b6 >= 0 goto line #139
129 ...
130 i7  = i1  + 1
131 i12 = i7  + 1
132 i17 = i12 + 1
133 i22 = i17 + 1
134 i27 = i22 + 1
135 i32 = i27 + 1
136 i37 = i32 + 1
137 i1  = i37 + 1   //all the 1's are the strides
138 goto line #127
```

Figure 3.8 The example of multiple strides. It is in the `OffHeapBitSet` class of Cassandra v2.0.8.

```
//Soot IR
530 $i5=r1.<ByteBuffer:limit>      //$i5 is the upper bound
531 if i1 >= $i5 goto line #536    //i1 is the loop index
532 $i9 = <STEP_LENGTH>;           //$i9 is the stride
533 i1 = i1 + $i9
    ...
535 goto line #531
```

Figure 3.9 The example of the stride is assigned outside of the function `total` where the loop resides. It is in the `CounterContext` class of Cassandra v2.0.8. The stride `STEP_LENGTH` is a static variable, which is assigned with 34 in the class initializer.

in the `OffHeapBitSet` class in Cassandra. The variable `i1` is the loop index while the variable `i0` is the upper bound. The loop index `i1` is updated multiple times from line #130 to line #137. DScope recursively applies all the assignments from line #130 to line #137 to get `i1 = i1 + 8`, and extracts the aggregated stride (i.e, 8). Figure 3.9 shows an example where the stride variable `$i9` is updated by `STEP_LENGTH` in the `CounterContext` class in Cassandra. The loop resides in the function `total()` while `STEP_LENGTH` is a static variable defined in the class initializer. DScope performs data flow analysis to extract the value of `STEP_LENGTH` from the class initializer and then checks whether the stride is positive because the loop index has an upper bound.

We now describe how to perform false positive filtering when the loop index, stride and bounds are denoted by the APIs in 60 commonly used Java classes and interfaces, listed in Table 3.1.

We classify those APIs into five categories, shown by Table 3.2: 1) the APIs which move the index forward when the Java class/interface has an upper bound; 2) the APIs which move the index backward when the Java class/interface has a lower bound; 3) the APIs which reset the index; 4) the APIs which check bounds; and 5) the APIs which update bounds. The APIs' names ending with "*" denote those APIs which perform similar operations and share the same prefix in their names. For example, in the `InputStream` family, there are `read()` and `readLine()` functions which both perform read operations on the corresponding `InputStream`.

DScope first extracts all the invoked APIs for each of the 60 Java classes and interfaces in the loop paths, and then prunes false positive candidates by checking whether 1) the "forward index" APIs or the "reverse index" APIs are invoked; 2) the "reset index" APIs and "update bounds" APIs are

Table 3.2 The APIs that are related to loop stride and bound update in 60 commonly used Java classes and interfaces. "*": a set of APIs perform similar operations; and "-": does not contain the corresponding type APIs.

Class	The type and name of APIs				
	Forward index	Reverse index	Reset index	Check bounds	Update bounds
File	create*	get*	new	is*, can* exists, get*	-
InputStream & Reader family	read*	reset	new	read*	new
DataInput family	read*	-	new	read*	new
Buffer family	position get* put*	position reset clear	duplicate allocate new	has* remaining	flip limit clear new
Channel family	read write	-	-	read write	-
List & Set	-	remove	new	is*	add clear new
Queue	-	poll remove	-	poll remove	add offer new
Stack	-	pop	-	empty pop	push new
Iterator & Enumeration	next	-	iterator elements new	has*	new
StringTokenizer	next	-	new	has*	new

not invoked; and 3) the "check bounds" APIs are invoked in the exit conditions. Note that, DScope cannot prune the case where both the "forward index" APIs and the "reverse index" APIs are invoked in the loop paths because the loop stride cannot be guaranteed to be *always* positive or negative.

The APIs in the five categories do not necessarily change the loop index or bounds. Those APIs' arguments should also be considered when DScope performs the false positive filtering. For example, `ByteBuffer` contains overloading methods which have an attribute called *relative* or *absolute*. The `ByteBuffer.get()` is a relative method while the `ByteBuffer.get(int)` is an absolute method. Invoking a relative method can change the loop index (i.e., `ByteBuffer.position`) while invoking absolute methods cannot. Another example is the `InputStream` class. Invoking the `InputStream.read(byte[], int, int)` with a zero size byte array or with 0 as the third parameter cannot change the loop index.

To achieve accurate pruning, DScope first annotates all the commonly used Java APIs with the attribute *change-positive*, *change-negative* or *change-possible*. The positive APIs can change the loop index (or bounds). The negative APIs cannot change either one. The possible APIs can possibly change the loop index (or bounds). For positive APIs, DScope's pruning steps are the

```
//DFSOutputStream.java                                          #HDFS-5438(v0.23.0)
1665 private void completeFile(ExtendedBlock last) throws IOException {
       ...
1667   boolean fileComplete = false;
1668   while (!fileComplete) {
1669    fileComplete = dfsClient.namenode.complete(src,dfsClient.clientName, last);
       ...
1689   }}
```

Figure 3.10 The code snippet of the HDFS-5438 bug. When the `ExtendedBlock last` is corrupted, the `fileComplete` variable is never set to be true, causing an infinite loop in DFSOutputStream.

same. For negative APIs in the type of "forward index", "reverse index", "reset index" or "update bounds", DScope ignores them when performing the pruning. For possible APIs, DScope performs intra-procedural data flow analysis on their parameters to decide whether these APIs change loop index/bounds or not.

DScope's false positive filtering only considers the commonly used Java APIs. If the loop index, stride or bounds are *only* related to specific application functions, which means the loop paths do not invoke any Java APIs in Table 3.2, DScope skips analyzing the loop and simply considers it as a false positive — this design decision may introduce false negatives, but greatly help the efficiency and accuracy of DScope.

One false negative example is the HDFS-5438 bug, shown by Figure 3.10. This hang bug is caused by a corrupted block, i.e, `last`. `DFSOutputStream` keeps polling `NameNode` to check the completeness of the committing block operation (line #1669). When the `last` block is corrupted, `NameNode` fails to commit it to the disk but returns `false` instead. This results in an infinite loop (line #1668-1689) causing a software hang in `DFSOutputStream`. DScope prunes this case because the loop paths do not invoke any Java APIs in Table 3.2. In fact, the loop path only invokes a specific application function, i.e., `complete()`. DScope should be able to detect this bug after adding inter-procedural analysis, which is however beyond the scope of this work.

3.3 Data Corruption Hang Bug Types

This section summarizes common types of corruption-hang bugs based on the detection results of DScope (the details of all the bugs detected by DScope will be presented in §4.3). Although DScope design was **not** affected by these types, this categorization can help future work on avoiding, detecting, and fixing corruption-hang bugs, and help developers better understand the impact of data corruption and corruption-hang bugs.

Our categorization is along two dimensions:

- What is the cause — is it specific error code returned by data operations (Type 1), or specific corrupted data content (Type 2), or specific exception thrown by data operations (Type 3, Type 4)?

- How did the cause lead to an infinite loop — is it through a direct data assignment (Type 1) or

```
//IOUtils.java                                              #Hadoop-8614(v0.23.0)
183 public static void skipFully(InputStream in, long len) throws IOException {
184  while (len > 0) {
185   long ret = in.skip(len); /* in is corrupted */
186   if (ret < 0) {           /* ret = 0 */
187     throw new IOException(...);
188   }
189   len -= ret;
    }}
```

Figure 3.11 The example when **error code returned by I/O operations directly impacts the loop stride**. Data corruption causes the I/O function, `InputStream.skip` returns 0, and 0 is used as the stride.

```
//BenchmarkThroughput.java                                  #HDFS-13514(v2.5.0)
172 public int run(...) throws IOException {
190  Configuration conf = getConf();/* conf is corrupted */
     ...
194  BUFFER_SIZE = conf.getInt(...);/* BUFFER_SIZE = 0 */
     ...
229 }
```

```
78  private void readLocalFile(Path path, ...) throws IOException {
     ...
83   InputStream in = new FileInputStream(...);
84   byte[] data = new byte[BUFFER_SIZE];
85   long size = 0;
86   while (size >= 0) {  /* size = 0 */
87    size = in.read(data);
    }}
```

Figure 3.12 The example when **corrupted data content indirectly impacts the loop stride**. The corrupted configuration file causes "BUFFER_SIZE = 0", which in turn makes the `InputStream in` perform read operation on a zero-size byte array and return 0. The loop's exit condition become infeasible because "size < 0" is never satisfied.

control-flow change (Type 3), or indirect data and control flow (Type 2, Type 4)?

Type 1: Error codes returned by I/O operations directly affect loop strides. For this type, the loop stride is directly assigned with a return value of an I/O operation. An infinite loop occurs when an unexpected error code is returned due to underlying data corruption. For example, as shown by Figure 3.11, when the log file (`InputStream in` at line #183) is corrupted due to bad encoding (Yarn-2724) or corruption propagation (Yarn-7179), the `InputStream in` becomes null. The `skip()` function returns 0 instead of the EOF indicator -1 (line #185). The return value `ret` is then used as the stride at line #189, which makes `len` never get updated but always stay larger than the lower bound (`len > 0`). As a result, the `skipFully()` function causes the system to hang by spinning in the loop forever. Variations of this hang bug type include the cases where the stride is always negative when the loop exit condition contains an upper bound or the stride is always positive when the loop exit condition contains a lower bound.

Type 2: Corrupted data content indirectly affects loop strides. This type of bugs occur when a specific piece of data is corrupted to certain unexpected values. Those values will then affect loop strides through data and/or control flow propagation and lead to infinite loops. For example,

```
//CombineFileInputFormat.java                                  #Mapreduce-2185(v0.23)
477 private static class OneFileInfo {
     ...
544   for (OneBlockInfo oneblock : blocks) {
545    blockToNodes.put(oneblock, oneblock.hosts);
       ...                              /*corrupted block's racks.length is 0*/
549    for (int j = 0; j < oneblock.racks.length; j++) {
550     String rack = oneblock.racks[j];
        ...
554     rackToBlocks.put(rack, blklist);
        ...
    }}}
```

```
255 private void getMoreSplits(...) throws IOException {
     ...
348   while (blockToNodes.size() > 0) {
       ...
359    for (Iterator<...> iter = rackToBlocks.entrySet().iterator();
360         iter.hasNext();) {
361     Map.Entry<...> one = iter.next();
        ...
363     List<OneBlockInfo> blocks = one.getValue();
        ...
369     for (OneBlockInfo oneblock : blocks) {
370      if (blockToNodes.containsKey(oneblock)){
371        blockToNodes.remove(oneblock);
          ...
    }}}}}
```

Figure 3.13 The example when **corrupted data content indirectly impacts the loop stride**. Data corruption causes `blockToNodes` and `rackToBlocks` to be different on the dimension of the blocks' number. This difference makes the corrupted block never been removed from the `blockToNodes` (i.e., zero-stride), causing the loop's exit condition to be infeasible. This is because "`blockToNodes.size() <= 0`" is never satisfied.

```
//TestProcfsBasedProcessTree.java                                  #Yarn-6991(v0.23.0)
//Thread #1
62 private class RogueTaskThread extends Thread {
63  public void run() {
64   try {
       ...
72     args.add(" echo $$ > " + pidFile + ";");
73     shexec = new ShellCommandExecutor(args...);
74     shexec.execute();------------------------>   Throw
       ...                                          Exception
79   } catch (IOException ioe) {
80    LOG.info("Error executing cmd");<----------
   }}}    /*file creation silently failed*/
```

```
//Thread #2
87 private String getRogueTaskPID() {
88  File f = new File(pidFile);
89  while (!f.exists()) {
       ...
91     Thread.sleep(500);
       ...
   }}
```

Figure 3.14 The example when **improper exception handling directly impacts the loop stride**. `ShellCommandExecutor.execute()` causes IOException. The exception is simply logged, and the creation of the `pidFile` is silently failed (i.e., zero-stride), which makes `File.exists()` always be false. "`-->`" represents the control flow.

as shown by Figure 3.12, when a configuration file (`conf` at line #190) is corrupted, the variable `BUFFER_SIZE` read from `conf` becomes 0. Calling `read()` function on a zero-size byte array at line #87 causes the loop stride to be zero and zero is then returned, which makes the loop's exit condition (`size < 0`) never be satisfied.

Figure 3.13 shows an example when the loop stride is indirectly impacted by the corrupted data content which involves *multiple* I/O related variables. The `blockToNodes` and `racktoBlock` are two maps which store different metadata information about every data block. If everything works correctly, these two maps should contain information about exactly the same set of blocks (i.e., every record in the `blocks` on line #544). However, if a block (`oneblock` at line #549) is corrupted and its `racks.length` becomes 0, this block will still be inserted into `blockToNodes` at line #545, but not be put into the `rackToBlocks` map with line #554 skipped. This would eventually cause the `while` loop on line #348 to hang. The reason is that this while loop keeps iterating until every block in `blockToNodes` is removed. Unfortunately, since only blocks that also exist in `rackToBlocks` map can be removed (line #369 – #371), the corrupted block will never be removed from `blockToNodes` and cause an infinite loop.

Type 3: Improper exception handling directly affects loop strides. Sometimes, a data-related operation itself is expected to update the loop stride. When this operation throws an exception, an improper exception handling may give up the operation, together with the associated stride updates, causing infinite loops. For example, the Yarn-6991 bug belongs to this type, shown by Figure 3.14. The `ShellCommandExecutor.execute()` function is expected to create a `pidFile`, whose existence

```
//Scrubber.java                                              #Cassandra-9881(v2.0.8)
 44 private final RandomAccessReader dataFile;
    ...
103 public void scrub(){
     ...
120  while (!dataFile.isEOF()){
      ...
129   try{                    /*dataFile is corrupted*/
130    key = sstable.partitioner.decorateKey(  //key is null
131     ByteBufferUtil.readWithShortLength(dataFile));
       ...
134    dataSize = dataFile.readLong();//skipped
       ...                                          Throw
139   } catch (Throwable th){                       Exception
140    throwIfFatal(th);//ignore Exception<- - -
141   }
      ...
185   try{
186    if (key == null)
187     throw new IOError(...);
       ...
207   } catch (Throwable th) {
208    throwIfFatal(th);//ignore IOError
       ...
    }}}
```

Figure 3.15 The example when **improper exception handling indirectly impacts the loop stride**. Data corruption causes the I/O function `decorateKey()` to throw exception at line #130-131, which makes the loop skip the index updating statement (i.e., zero-stride) at line #134. "- - >" represents the control flow.

will help a while loop (line #89) to exit. When the disk is full, `ShellCommandExecutor. execute()` throws an exception at line #74. This exception is simply logged. Consequently, without the creation of `pidFile`, the while loop at line #89 never exits.

Type 4: Improper exception handling indirectly affects lo- op strides. For this type of bugs, the stride-update operation itself did not raise any exceptions. However, an exception handling of another operation, a data-related operation, changes the control flow and causes the stride update to be skipped. For example, the Cassandra-9881 bug matches this type, shown by Figure 4.8. When the `dataFile` (`RandomAccessReader` at line #131) is corrupted, the `decorateKey()` function cannot recognize it, thus throws an exception without assigning `key` at line #130 (i.e., `key == null`), or executing `dataFile.readLong()` at line #134. But this exception is simply ignored because it's not fatal at line #140. When the `key` is null, the `scrub()` function throws an `IOError` (line #187), catches it (line #207), and ignores it because it's not a fatal error (line #208). Without moving the index (i.e., zero-stride) by calling `dataFile.readLong()` at line #134, the `scrub()` function keeps reading from the same place, looping forever.

Discussion Theoretically, other types of corruption-hang bugs could exist, like corruption affecting loop bounds, instead of loop strides, or corrupted data content directly, instead of indirectly, affects loop strides. DScope bug detection algorithm *can* detect those types of bugs too. However, we did not observe them in the real-world bugs that we have encountered.

Table 3.3 The cloud server systems used in our evaluation and the number of detected data corruption hang bugs in each system.

System	Description	# of bugs
Cassandra	Distributed database management system	2
Compress	Libraries for I/O ops on compressed file	2
HD Common	Hadoop utilities and libraries	10
Mapreduce	Hadoop big data processing framework	5
HDFS	Hadoop distributed file system	4
Yarn	Hadoop resource management platform	4
Hive	Data warehouse	12
Kafka	Distributed streaming platform	1
Lucene	Indexing and search server	2
Total		42

3.4 Evaluation

In this section, we present our experimental evaluations on DScope. We first describe our evaluation methodology and then discuss our evaluation results in detail.

3.4.1 Evaluation Methodology

DScope is implemented on top of Soot v2.5.0 [Soo], a Java bytecode analysis infrastructure, with the latest Coffi library [Ver96], written in Java language with about 18,000 lines of code. Our experimental evaluation covers a wide range of popular cloud server systems listed in Table 3.3: Cassandra is a distributed key-value store; Compress provides libraries for I/O operations on compressed files; Hadoop common provides utilities and libraries for all Hadoop projects; Hadoop MapReduce is a big data processing platform; HDFS is a distributed file system; Hadoop Yarn is a distributed resource management service; Hive is a data warehouse; Kafka is a distributed streaming system; and Lucene is a data indexing and searching server. We try to cover as many cloud server systems as possible to show that data corruption hang bugs are widespread in the real world.

All the experiments were conducted in our lab machine with an Intel® Xeon® E5-1630 Octa-core 3.7GHz CPU, 16GB memory, running 64-bit Ubuntu 16.04 with kernel v4.13.0. Our evaluation considers both coverage (i.e., true positives) and precision (i.e., false positives) of data corruption hang bug detection. We also compare DScope with two state-of-the-art static bug detection tools, Findbugs(v3.0.1) [Fin] and Infer(v0.9.2) [Fbi].

For all the hang bugs reported by DScope, we first manually validate them by checking whether we can reproduce the software hang symptom after injecting data corruption into the corresponding data. We first check DScope's analysis results to identify which faulty I/O operations affect the loop strides. We then inject the faults (e.g., corrupted data content, corrupted configuration files, disk exhaustion) into the corresponding I/O operations. If the software hang does happen, we mark the bug as a true positive. Otherwise, we consider it as a false positive. For all the true positives, we

Table 3.4 The detection comparison of DScope with Findbugs and Infer on all the 9 systems. "TP": the number of true positive bugs by each scheme; "FP": the number of false positive bugs reported by DScope; "-": runtime execution errors (Infer).

System		Release date	DScope		Findbugs	Infer
			TP	FP	TP	TP
Cassandra	v2.0.8	2014/05/29	2	1	0	1
Compress	v1.0	2009/05/21	2	2	0	-
HD Common	v0.23.0	2011/11/11	4	6	0	0
	v2.5.0	2014/08/11	6	6	0	0
Mapreduce	v0.23.0	2011/11/11	3	0	0	0
	v2.5.0	2014/08/11	2	0	0	0
HDFS	v0.23.0	2011/11/11	1	1	0	0
	v2.5.0	2014/08/11	3	5	1	-
Yarn	v0.23.0	2011/11/11	2	2	1	0
	v2.5.0	2014/08/11	2	5	0	0
Hive	v1.0.0	2015/05/20	7	6	0	-
	v2.3.2	2017/11/18	5	1	0	0
Kafka	v0.10.0.0	2016/05/22	1	1	0	0
Lucene	v2.1.0	2007/02/17	2	1	0	0
Total			42	37	2	1

then search the bug repository (i.e., JIRA [Jir]) to see whether they are already reported. If they are, we mark them as the existing data corruption hang bugs. Otherwise, we report them in the bug repository and mark them as newly discovered data corruption hang bugs.

We then use the true positives detected by DScope as the benchmark to evaluate the detection efficacy of Findbugs and Infer. For these two tools, if they report at least one line of the code related to a data corruption hang bug (e.g, a line of the data corruption loop body or a line contains a variable which is then used in the loop), we consider the reported issue as a true positive. We omit the false positives of Findbugs and Infer in our evaluation because these two generic bug detection tools can report hundreds or thousands of suspicious issues. For example, Findbugs and Infer reports 5,434 and 13,993 issues in Hive v2.3.2, respectively. It is extremely time-consuming to validate all of their detection results manually. It is also wrong to label all the issues identified by Findbugs or Infer but not DScope as false positives since some of those issues are true bugs although they are not related to data corruption hang bugs.

3.4.2 Bug Detection and Precision Results

Table 3.4 and 3.5 show the detection results achieved by different sche- mes. DScope reports 79 data corruption hang bugs, with 42 of them being true bugs, and 29 out of the 42 bugs are newly discovered bugs. Note that, we ran DScope on the target cloud server systems and identified those 42 bugs. But it does not mean that those 42 bugs include *all* the data corruption bugs in those systems. There are some other types of data corruption hang bugs that we cannot identify. For example, data corruption causes the recursive functions never end, making system hang. However, it is out of the

Table 3.5 The detection comparision of DScope with Findbugs and Infer on all the 42 data corruption hang bugs.

#	Bug name	System version	Bug type	Known or new	Deteced		
					DScope	Findbugs	Infer
1	Cassandra-7330	v2.0.8	#1	known	✓	✗	✗
2	Cassandra-9881	v2.0.8	#3	known	✓	✗	✓
3	Compress-87	v1.0	#1	known	✓	✗	✗
4	Compress-451	v1.0	#2	new	✓	✗	✗
5	Hadoop-8614	v0.23.0	#1	known	✓	✗	✗
6	Hadoop-15088	v2.5.0	#1	new	✓	✗	✗
7	Hadoop-15415	v0.23.0	#2	new	✓	✗	✗
8		v2.5.0	#2	new	✓	✗	✗
9	Hadoop-15417	v0.23.0	#2	new	✓	✗	✗
10		v2.5.0	#2	new	✓	✗	✗
11	Hadoop-15424	v2.5.0	#1	new	✓	✗	✗
12	Hadoop-15425	v2.5.0	#1	new	✓	✗	✗
13	Hadoop-15429	v0.23.0	#2	new	✓	✗	✗
14		v2.5.0	#2	new	✓	✗	✗
15	HDFS-4882	v0.23.0	#3	known	✓	✗	✗
16	HDFS-5892	v2.5.0	#2	known	✓	✓	✗
17	HDFS-13513	v2.5.0	#2	new	✓	✗	✗
18	HDFS-13514	v2.5.0	#2	new	✓	✗	✗
19	Mapreduce-2185	v0.23.0	#2	known	✓	✗	✗
20	Mapreduce-2862	v0.23.0	#2	known	✓	✗	✗
21	Mapreduce-6990	v0.23.0	#1	new	✓	✗	✗
22	Mapreduce-7088	v2.5.0	#1	new	✓	✗	✗
23	Mapreduce-7089	v2.5.0	#1	new	✓	✗	✗
24	Yarn-163	v0.23.0	#1	known	✓	✓	✗
25	Yarn-2905	v2.5.0	#1	known	✓	✗	✗
26	Yarn-6991	v0.23.0	#4	new	✓	✗	✗
27		v2.5.0	#4	new	✓	✗	✗
28	Hive-5235	v1.0.0	#1	known	✓	✗	✗
29	Hive-13397	v1.0.0	#2	known	✓	✗	✗
30	Hive-18142	v1.0.0	#2	new	✓	✗	✗
31	Hive-18216	v2.3.2	#1	new	✓	✗	✗
32	Hive-18217	v2.3.2	#1	new	✓	✗	✗
33	Hive-18219	v1.0.0	#2	new	✓	✗	✗
34		v2.3.2	#2	new	✓	✗	✗
35	Hive-19391	v1.0.0	#2	new	✓	✗	✗
36	Hive-19392	v1.0.0	#2	new	✓	✗	✗
37		v2.3.2	#2	new	✓	✗	✗
38	Hive-19395	v1.0.0	#1	new	✓	✗	✗
39	Hive-19406	v2.3.2	#2	new	✓	✗	✗
40	Kafka-6271	v0.10.0	#1	new	✓	✗	✗
41	Lucene-772	v2.1.0	#2	known	✓	✗	✗
42	Lucene-8294	v2.1.0	#2	new	✓	✗	✗
Total #					42	2	1

```
//cassandra-2.0.8: Scrubber.java
 41 private boolean isCommutative = false;
    ...
103 public void scrub(){
     ...
120  while (!dataFile.isEOF()){
      ...
127   DecoratedKey key = null;
      ...
248   throwIfCommutative(key, th);//Infer: null parameter
      ...
    }}
```

```
327 private void throwIfCommutative(DecoratedKey key, Throwable th) {
328  if (isCommutative && !skipCorrupted){
329    outputHandler.warn(String.format("...", key));
       ...
    }}
```

Figure 3.16 Infer identifies a null parameter problem in the `throwIfCommutative()` function at line #248. The Cassandra-9881 bug happens at line #103-256.

scope of this thesis, which is part of our future work.

In contrast, existing generic bug detection tools cannot detect most of those 42 data corruption hang bugs. Findbugs only identifies the HDFS-5892 and Yarn-163 bugs while Infer only identifies the Cassandra-9881 bug. Those results are expected because no previous static analysis tools, including Findbugs and Infer, have targeted data corruption hang bugs. Findbugs targets bugs that follow specific anti-patterns in Java programs, such as "private method is never called", "method concatenates strings using + in a loop", and "unchecked type in generic call", none of which are related to data corruption hang bugs detected by DScope. Note that, Findbugs does contain one specific anti-pattern called "an apparent infinite loop" which is related to data corruption hang bugs. However, Findbugs only reports two suspicious issues on the target cloud server systems and both issues involve a `while(true)` type loop. After further inspection, these two loops can exit eventually due to timeouts. Infer mostly focuses on memory and resource leak bugs, and hence cannot detect most corruption-hang bugs shown in Table 3.5.

Findbugs identifies the HDFS-5892 bug, as it discovers `getFinalizedDir()` can be "null" in the loop body. This bug happens when corrupted data content indirectly affects the loop stride (i.e, the `getFinalizedDir().length` becomes 0). Indeed, the `getFinalizedDir()` function is called during the loop's execution, but it is not the root cause of this data corruption hang bug. Findbugs identifies the Yarn-163 bug, as it discovers that encoding the `InputStreamReader reader` to a `FileReader` can corrupt the `reader`, which is related to the data corruption hang bugs — performing skip operations on a corrupted `FileReader` can cause the `skip` function to return error code (i.e, 0).

Infer identifies the Cassandra-9881 bug, as it discovers that the `scrub()` function in the `Scrubber` class could invoke a `throwIfCommutative()` function at line #248 with null parameter, shown by Figure 3.16. As we discussed in §3.3, when data corruption happens, `key` fails to be assigned to new

Table 3.6 The types and the number of false positives pruned by DScope.

System		Pruned FP	
		Numeric primitives	**Java APIs**
Cassandra	v2.0.8	386	71
Compress	v1.0	147	20
HD Common	v0.23.0	1023	378
	v2.5.0	1650	790
Mapreduce	v0.23.0	377	363
	v2.5.0	938	641
HDFS	v0.23.0	312	323
	v2.5.0	1723	1073
Yarn	v0.23.0	151	214
	v2.5.0	451	665
Hive	v1.0.0	4268	3003
	v2.3.2	5269	3663
Kafka	v0.10.0.0	186	441
Lucene	v2.1.0	287	44
Total		17168	11689

values and sticks with the default value, "null". This makes the `scrub()` function skip updating the index, causing an infinite loop. Indeed, the `throwIfCommutative()` function is called during the loop's execution, but it is not the root cause of this data corruption hang bug. In fact, it does not break the loop to prevent `scrub()` from hanging. This is because the `isCommutative` variable is false, which makes the `if` branch at line #329 unreachable. Thus, even with a null parameter, the `throwIfCommutative()` can still execute successfully at line #248.

Table 3.5 also shows the types of the detected data corruption hang bugs. As we can see, "Type 1" and "Type 2" cover most of the detected bugs — 16 and 22 bugs respectively. This indicates that most of the data corruption hang bugs happen when the data corruption causes the error code returned by I/O operations to directly impact the loop stride or corrupted data content indirectly impacts the loop stride.

To understand how DScope does not prune all the false positives, we manually study those 37 false positives in Table 3.4. We find most of cases require inter-procedural analysis to identify. We will discuss it in §3.5.

As shown in Table 3.6, DScope prunes 28,857 false positives in total, including 17,168 cases where the loop index, stride and bounds are denoted by numeric primitives, and 11,689 cases where the loop index, stride and bounds are denoted by commonly used Java APIs.

We should note that, we do not intend to claim that DScope can replace those generic bug detection tools such as Findbugs and Infer. We believe our bug detection schemes are complementary to those existing tools and could be used in combination by the software developer.

3.5 Discussion

We observe that in most of the 37 false positive cases, the forwarding-index/reversing-index Java APIs and the checking-bounds Java APIs are located in different application functions. These APIs are indirectly invoked in the application functions which are invoked in the loop paths. To further reduce false positives, we plan to conduct inter-procedural analysis on all the bug candidates to generate the loop paths where the loop index, stride, and bounds are denoted by either numeric primitives or Java APIs. We then adopt DScope's false positive pruning principles to prune the false positives without missing true positives.

3.6 Summary

In this chapter, we have presented DScope, a new data corruption hang bug detection tool for cloud server systems. DScope combines candidate bug discovery and false positive pattern filtering to detect software hang bugs that are related to data corruptions. DScope is fully automatic without requiring any user input or pre-defined rules. We have implemented a prototype of DScope and evaluated it over 9 commonly used cloud server systems. DScope successfully detects 42 true corruption hang bugs (29 of them are new bugs) while existing bug detection tools can only detect very few of them (2 by Findbugs and 1 by Infer).

CHAPTER

4

HANGFIX: AUTOMATICALLY FIXING SOFTWARE HANG BUGS IN CLOUD SYSTEMS

4.1 Introduction

Open-source server systems such as Cassandra [Cas], HBase [Hba], Hadoop [Had], and Kafka [Kaf] have enabled many real-world applications ranging from social media [Mea09; Ins; Twi] to business intelligence [Sap; Mic; Sas] and security detection [Sno; Oss; Ste]. However, due to their inherent complexity, those server systems suffer from different performance and availability issues. Particularly, software hang bugs impose most severe impact to server systems, which often cause serious service outages. For example, in 2015, Amazon DynamoDB experienced a five-hour service outage [Dyn; Aws] affecting many AWS customers including Netflix, Airbnb, IMDb. The root cause of the service outage is a software hang bug where an improper error handling keeps sending new requests to overloaded metadata server, causing further cascading failures and retries. In 2017, British Airways experienced a serious service outage with a penalty of more than £100 millions [Bac] due to a software hang bug triggered by corrupted data during data center failover.

Unfortunately, software hang bugs are notoriously difficult to debug because they typically produce little diagnostic information. The problem exacerbates in the cloud environment due to the difficulty of reproducing the bug in the development environment. Although previous bug detection tools [Dai18a; He18; Liu14; Wan08; Cot09] can detect those hang bugs, production service outage

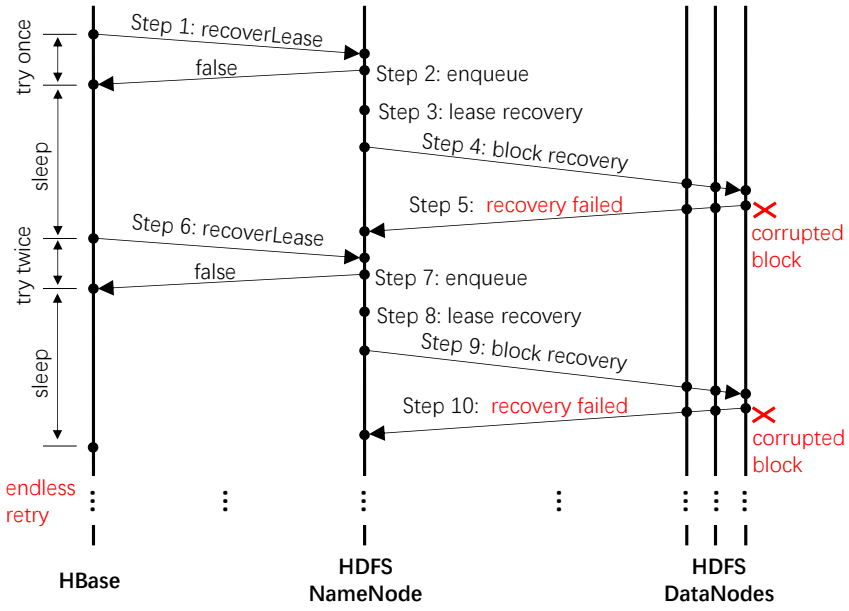

Figure 4.1 The HBase-8389 bug. HBase endlessly sends lease recovery requests to HDFS.

```
//FSHDFSUtils.java                                              HBase-8389(v0.94.3)
   + private String RECOVERY_TIMEOUT_KEY ="recover.timeout";
   + private int DEFAULT_RECOVER_TIMEOUT = 900000;
   + private long timeout = conf.getInt(RECOVER_TIMEOUT_KEY,
       DEFAULT_RECOVER_TIMEOUT);
 48  public void recoverFileLease(..., final Path p, ...) throws IOException {
      ...
 62   boolean recovered = false;
 63   int nbAttempt = 0;
   +  long st = System.currentTimeMillis();
 64   while (!recovered) {
 65    nbAttempt++;
       ...
 71    recovered = dfs.recoverLease(p);//send to HDFS
       ...
 85    if (!recovered) {
        ...
 96     Thread.sleep(nbAttempt < 3 ? 500 : 1000);
        ...
103    }
   +   long elapsed= System.currentTimeMillis() - st;
   +   if (timeout > 0 && elapsed >= timeout)
   +    throw new TimeoutException("Timed out. Breaking infinite polling!");
104 }}
```

Figure 4.2 When the blocks are corrupted, the `recoverLease()` function keeps polling the recovery results, getting "false" and sending new recovery request, hanging in an infinite loop. "+" means added code, representing the patch for this bug.

cannot be resolved until the hang bugs are fixed. Recent studies [Gun14; Dai18a; He18; Dea15a; Dai18c] have shown that many hang bugs are caused by unexpected runtime data corruptions or inter-process communication failures, which makes those hang bugs particularly difficult to diagnose and fix. Bug repository records indicate that those hang bugs often take days or even months to be resolved.

4.1.1 A Motivating Example

We use the HBase-8389 bug as one example shown by Figure 4.1 to illustrate how the data corruption and infinite polling trigger the hang bug and how the hang bug affects the cloud service. This bug was triggered when HBase sends a `recoverLease` request to the NameNode for recovering some data blocks. NameNode responds to HBase instantaneously with a `false` reply and enqueues the request for later processing to achieve low-latency asynchronous communication. After getting the `false` reply, HBase blocks itself by sleeping for a period of time. When NameNode processes the request, it recovers the lease and sends block recovery requests to the DataNodes. In the case of data corruption, DataNodes abort the recovery operation and sends the `recovery failed` message to the NameNode which will in turn send a `false` reply to HBase. After waking up from sleep, HBase resends the `recoverLease` request to the NameNode which repeats the recovery procedure and keeps failing. When this hang bug is triggered, HBase becomes blocked for all the other requests and HDFS keeps performing the same recovery operations for corrupted data. This bug is difficult to debug because HBase does not produce any log information and HDFS provides many misleading error messages. It took the developer 24 days to fix the bug after submitting 10 versions of unsuccessful patches.

Figure 4.2 shows the buggy code snippet and the patch. HBase inquires the previous recovery results of p from HDFS at line #71. After getting an unsuccessful recovery reply, HBase sleeps for 1 second or less at line #96 and retries endlessly in a while loop between lines #64-104. This HBase hang bug is caused by the infinite polling loop and the final patch created by the developer is to add a timeout check mechanism and throws an timeout exception to break out of the infinite polling loop.

4.1.2 Our Contribution

This chapter presents HangFix, a software hang bug fixing tool that can automatically recovers a system from a software hang bug. We focus on recovering software hang problems caused by infinite loops or endless blocking operations. For example, a software hang occurs when the program enters an infinite loop where the loop stride is incorrectly updated. HangFix first performs hang bug pattern recognition by analyzing the application byte code and the hang functions. Based on the identified hang bug patterns, HangFix then produces corresponding patches by leveraging existing exception handling mechanisms to break out of an infinite loop or blocking state. HangFix is application-agnostic, which does not require any application source code or knowledge to produce the hang bug fix.

Specifically, this chapter makes the following contributions:

- We present an *application-agnostic* hang bug fixing scheme that can automatically correct software hang problems and produce patched application byte code.
- We extract commonly seen software hang bug patterns and provide corresponding automatic patching strategies.

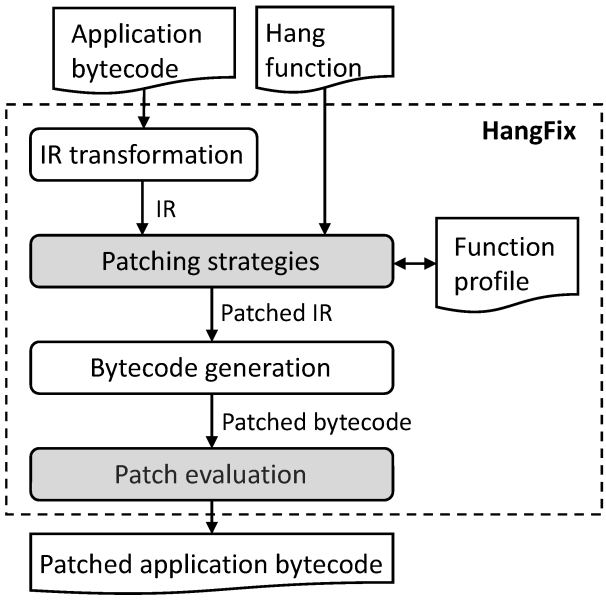

Figure 4.3 The architecture of HangFix.

- We have implemented a prototype of HangFix and evaluated our system using 42 real-world software hang bugs in 10 commonly used cloud server systems (e.g., Cassandra, HDFS, Mapreduce, HBase, etc). HangFix automatically fixes 40 of them in seconds.

4.2 System Design

This section first provides an overview of HangFix. We then describe each hang bug pattern and its corresponding fix strategies in detail.

4.2.1 System Overview

HangFix aims at providing application-agnostic software hang bug fix, which can automatically recover a server application from a detected hang problem. HangFix focuses on correcting two commonly seen root causes of hang bugs: 1) the program enters an infinite loop due to incorrectly updated loop stride or index triggered by data corruptions; and 2) a blocking operation makes the program wait for a state to be reached indefinitely due to unexpected inter-process communication failures.

HangFix takes the hang function detected by existing hang bug detection tools (e.g., [Liu14; Wan08; Car11; Dai18a]) and application byte code as inputs and leverages static program analysis techniques to identify hang bug patterns and bug triggering points. It then inserts code patches at the triggering point to make the program break out of infinite loops or indefinite waiting state. Figure 4.3 shows the overall structure of the HangFix system. Specifically, HangFix can detect four commonly seen hang bug patterns: 1) unexpected return value causes the loop stride to be incorrectly updated, 2) misconfigured variables cause the loop stride or index incorrectly updated, 3) improper exception

```
//StreamReader.java                                          Cassandra-7330(v2.0.8)
 73 public SSTableWriter read(ReadableByteChannel channel) throws IOException {
      ...
 81   DataInputStream dis = new DataInputStream(new LZFInputStream(Channels.
    newInputStream(channel)));
      ...
 96   drain(dis, in.getBytesRead());
      ...
102 }

114 protected void drain(InputStream dis, long bytesRead) throws IOException {
115   long toSkip = totalSize() - bytesRead;
116   toSkip = toSkip - dis.skip(toSkip);
117   while (toSkip > 0) {
118-   toSkip = toSkip - dis.skip(toSkip);
    +   long skipped = dis.skip(toSkip);
    +   toSkip = toSkip - skipped;
    +   if (skipped <= 0) {
    +    throw new IOException("Unexpected return value causes the loop stride"
    +                                          +"to be incorrectly updated.");}
119 } }
```

Figure 4.4 Example of hang bug pattern #1 and its fixing strategy. When `InputStream dis` is inaccessible or corrupted by bad encoding `dis.skip` can return -1 or 0, and -1/0 is used as the stride. "⟶" represents the function call invocation. "-" means deleted code and "+" means added code, representing the patch generated by HangFix.

or error handling skips loop index updating operations, and 4) blocking-prone operations. HangFix currently focuses on Java programs and leverages Soot compiler's transformation module [Soo] to translate bytecode into intermediate representation (IR) code. It starts from the hanging function, traverses its control flow graph (CFG), conducts data-dependency, intra- and inter-procedural analysis, checks whether the target problem matches our hang bug patterns.

To correct a software hang bug safely, HangFix inserts a known exception throwing statement at the identified bug triggering points and adds hang fix logic in the exception handling procedure. For example, HangFix can ignore unexpected return values to break out of loop or revert a misconfigured variable value that triggers the infinite loop. By leveraging known exception handling, HangFix avoids introducing new semantics into the code patch while restoring the program execution flow. HangFix leverages Soot transformation module to generate the new application byte code from the patched IR code. After patching, we evaluate the generated patch to check whether the program executes successfully without hang or crash when the bug is triggered.

4.2.2 Fixing Hang Bugs Related to Unexpected Return Values

Hang bug pattern #1: Unexpected return value causes the loop stride to be incorrectly updated.

In this class of hang bugs, the loop stride depends on an operation's returning value and the operation returns an unexpected error code due to some underlying faults such as data corruption. Figure 4.4 shows an example falling into this bug pattern—the Cassandra-7330 bug. When `InputStream dis` read from a channel at line #81 is inaccessible or corrupted by bad encoding, the `skip()` function returns -1 or 0 instead of any positive number (line #118). The return value is

```
//ZipArchiveInputStream.java                                    Compress-87(v1.0)
255 public long skip(long value) throws IOException {
257  long skipped = 0;
258  byte[] b = new byte[1024];
259  while (skipped != value) {
260   long rem = value - skipped;
261   int x = read(b, 0,
262           (int)(b.length > rem ? rem : b.length));
   +   if(x == 0) {
   +    throw new IOException("Unexpected return value causes the loop stride"
   +                                          +"to be incorrectly updated.");}
263   if (x == -1)  return skipped;
264   skipped += x;
265  }
     ...
270 }
```

Figure 4.5 Example of hang bug pattern #1 and its fixing strategy. Reading on a truncated archive, can return 0 at line #261, and 0 is used as the stride. "+" means added code, representing the patch generated by HangFix.

then used as the stride to be subtracted from `toSkip` at line #118, which makes `toSkip` always get increased or never get updated, thus always be larger than the lower bound (`toSkip > 0`). As a result, the `drain()` function causes the system to hang by spinning in the loop forever.

Figure 4.5 shows another example of this bug pattern that exists in a different system. When `ZipArchiveInputStream` is corrupted due to a truncated archive, conducting `read()` operation on the corrupted `ZipArchiveInputStream` at line #261 returns 0. The return value is used as the stride to be added to `skipped` at line #264, which makes `skipped` never get updated, thus never be equal to `value` at line #259. As a result, the `skip()` function causes the system to hang.

Patching strategy #1: Checking error-prone return values to terminate an infinite loop.

To fix hang bugs falling into our first hang bug pattern, we first identify those error-prone return values which are used to update the loop stride and then add proper checkers over those return values. In the case of unexpected errors, we terminate the loop by throwing an exception with a known type, that is, the same type of exception declared in the hang function signature. Specifically, HangFix traverses the CFG of the hang function f, extracts all the invocation statements from f and checks whether every error-prone return value of every invocation is checked. HangFix leverages static analysis to identify those error-prone function return values. Specifically, we study all the commonly used Java library functions, which are related to loop stride, index, or bounds to create a *function profile* whose return values are error-prone.

If an error-prone return value r_{err} from the function f_i is not checked properly before the loop or along the loop path, HangFix inserts a checker after the corresponding invocation statement of f_i. The checker is an `if` branch, with the condition of $r_i == r_{err}$. If function f_i has more than one error-prone return values, HangFix generates a combined condition in the form of $r_i >= r_{err_{min}}$ or $r_i <= r_{err_{max}}$. Inside the `if` branch, a known exception is thrown with error message in the format of "underlying failure causes function f_i returns r_{err}, affecting loop stride, leading to an infinite loop". This known exception's type is the same as the type of the exception declared by function f.

If function f does not contain "throws exception" clause in its signature, HangFix checks the call stack of f backwards until it identifies the n-hop caller function of f who declares a checkable exception in its function signature. HangFix then inserts the same checkable exception in the signatures of function f and its i-hop callers, $i = 1, 2, ..., n-1$. If there are more than one checkable exceptions, HangFix chooses the first and most specific one.

If function f such as `Thread.run()` cannot throw checkable exceptions, HangFix introduces a flag variable in its immediate caller function, sets the flag when exception happens in f and checks the flag in the caller. HangFix then propagates the failure by adding the "throws exception" clause in the caller's function signature.

For example, the Cassandra-7330 bug in Figure 4.4 can be fixed using this patching strategy. Under abnormal conditions (e.g., inaccessible or corrupted `InputStream`), conducting `skip()` operation can return unexpected values (i.e., 0 or -1). HangFix first identities those return values and detects that function `drain()` does not contain any checking over those error-prone return values. HangFix then inserts a checker after the invocation of `InputStream.skip()` which includes introducing a local variable `skipped` to store the return value, generating a combined error-prone value (i.e., <= 0), comparing `skipped` with <= 0, and throwing an `IOException` with error message "underlying failure causes function skip returns <= 0, affecting loop stride, leading to an infinite loop". The Compress-87 bug in Figure 4.5 can be fixed in a similar way.

4.2.3 Fixing Hang Bugs Related to Misconfigured Variables

Hang bug pattern #2: Misconfigured variables cause the loop stride or index incorrectly updated.

This category of bugs occur when a parameter of an operation or a class field variable is misconfigured. Those values can either reset the loop index to the same value in each iteration, or set the loop stride to be non-positive (or non-negative) infinitely when the loop has a fixed upper (or lower) bound.

For example, Figure 4.6 shows a Hadoop bug example where the `bufferSize` parameter is misconfigured to be 0 at line #97 and passed in as an argument at line #74. `InputStream in` performs the `read()` operation on a zero-size byte array and returns zero at line #84. The returned zero is then assigned to the loop index, i.e., `byteRead`, making the loop's exit condition (`byteRead < 0`) never be satisfied. As a result, the `copyBytes()` function endlessly spins in the loop. Figure 4.7 shows another example, where the class field variable `BUFFER_SIZE` is misconfigured to be 0 or negative at line #194, which affects the loop stride and makes the loop index `size` always get increased or never get updated at line #70. As a result, the `writeLocalFile()` function hangs endlessly.

Patching strategy #2: Adding missing check over misconfigured variable to restore default value or break the loop if the default value does not exist.

Different from the patching strategy #1 which disrupts a loop iteration by throwing a checkable and acceptable exception, this patching strategy identifies the misconfigured variables after inquiring the *function profile* and 1) automatically restores the default configuration values or 2) breaks

```
//IOUtils.java                                                    Hadoop-15415(v2.5.0)
 96 public static void copyBytes(InputStream in, ..., Configuration conf) throws
     IOException {
 97   int buffSize = conf.getInt("io.file.buffer.size",4096);
   + if(buffSize == 0)
   +   buffSize = 4096; //reassigned to default
 98   copyBytes(in, ..., buffSize, true);
 99 }

49 public static void copyBytes(InputStream in, ..., int buffSize, boolean close)
     throws IOException {
       ...
52     copyBytes(in, ..., buffSize);
       ...
65 }

74 public static void copyBytes(InputStream in, ..., int buffSize) throws
     IOException {
       ...
77   byte[] buf = new byte[buffSize];
78   int bytesRead = in.read(buf);
79   while (bytesRead >= 0) {
       ...
84     bytesRead = in.read(buf);
   +   if(buffSize == 0) break; //at loop tail
85 }}
```

Figure 4.6 Example of hang bug pattern #2 and its fixing strategy. Misconfiguration causes `bufferSize` to be 0, which in turn makes the `InputStream in` perform `read` operation on a zero-size byte array and return 0. "⟶" represents the function call invocation, while "⇢" represents the data dependency flow. "+" means added code, representing the patch generated by HangFix.

```
//BenchmarkThroughput.java                                        HDFS-14481(v2.5.0)
 52 private int BUFFER_SIZE; //class field

172 public int run(...) throws IOException {
       ...
194   BUFFER_SIZE = conf.getInt("buffer.size", 4*1024);
    + if(BUFFER_SIZE <= 0)
    +   BUFFER_SIZE = 4096; //reassigned to default
       ...
229 }

 63 private Path writeLocalFile(..., long total) ...{
       ...
 70   for(long size = 0; size < total; size += BUFFER_SIZE){
         ...
     }}
```

Figure 4.7 Example of hang bug pattern #2 and its fixing strategy. The class field `BUFFER_SIZE` is misconfigured to be 0 or negative at line #194, which in turn makes the `size` never get updated at line #70. "⇢" represents the data dependency flow. "+" means added code, representing the patch generated by HangFix.

the loop if the variable's default value does not exist.

Specifically, HangFix traverses the CFG of the hang function f, extracts all the invocation statements from f, inquires the *function profile*, and checks whether every error-prone parameters from every invocation is checked. If there is no invocation statements in f, HangFix considers the stride variable is error-prone with non-positive (or non-negative) error value when the loop has a

fixed upper (or lower) bound. If an error value v_{err} for the variable v is not checked properly before the loop or along the loop path, HangFix inserts a checker to reassign the misconfigured variable with its default value.

The checker is an `if` branch with condition of $v == v_{err}$. If v has more than one error values, HangFix generates a combined condition in the form of $v >= v_{err_{min}}$ or $v <= v_{err_{max}}$. The checker is inserted after the configuration statement. HangFix conducts inter-procedural analysis on the whole program to retrieve the call graph to function f and then performs data-dependency analysis backwards along the call path to identify the configuration statement.

If the configuration statement cannot be identified or the hang function f is public which means it can be directly accessed by user-defined classes or classes in other integrated systems, HangFix inserts another checker to break the loop. This checker is an `if` branch, inside which, there is a `break` statement. The checker is inserted at the loop tail.

For example, the Hadoop-15415 bug in Figure 4.6 can be fixed using this patching strategy. The `byte[] buf` in `InputStream.read(buf)` operation at line #84 is an error-prone parameter. HangFix first inquires the *function profile* to find that `buf` is error-prone when its size is 0. HangFix then conducts data-dependency analysis to identify the local variable of `buf.size` to be `buffSize` at line #77. After retrieving the call graph to `copyBytes()` function at line #74, HangFix identifies the configuration statement for `buffSize` at line #97. HangFix then inserts a checker after line #97, including an `if` branch with condition `buffSize == 0` and an assignment statement of `buffSize = 4096`, where 4096 is the default value read from the configuration statement. Since the `copyByte()` function at line #74 is public, it can be accessed by user-defined classes or classes in any other integrated systems (e.g., HBase, Hive, etc). The argument value of `buffSize` passed in from those classes are unaccessible in our analysis currently. Thus, another checker inside the `copyByte()` function is necessary. At the loop tail, between line #84 and #85, HangFix inserts an `if` branch with condition `buffSize == 0` and a `break` statement.

The HDFS-14481 bug can also be fixed using this strategy. HangFix identifies the class field variable `BUFFER_SIZE` is error-prone with non-positive error values because it is used as the loop stride in the "`size += BUFFER_SIZE`" clause at line #70. HangFix conducts data-dependency analysis to identify the configuration statement for `BUFFER_SIZE` at line #194. HangFix inserts a checker after line #194, including an `if` branch with condition `BUFFER_SIZE <= 0` and an assignment statement of `BUFFER_SIZE = 4096`, where 4096 is the default value. In this case, HangFix does not need to insert a second checker at the loop tail because the loop stride is only affected by the class field variable `BUFFER_SIZE`. Thus, checking `BUFFER_SIZE` after the configuration statement is sufficient.

4.2.4 Fixing Hang Bug Related to Improper Exception or Error Handling

Hang bug pattern #3: Improper exception or error handling skips loop index updating operations.

This category of hang bugs occur when exception happens or error code is returned and an im-

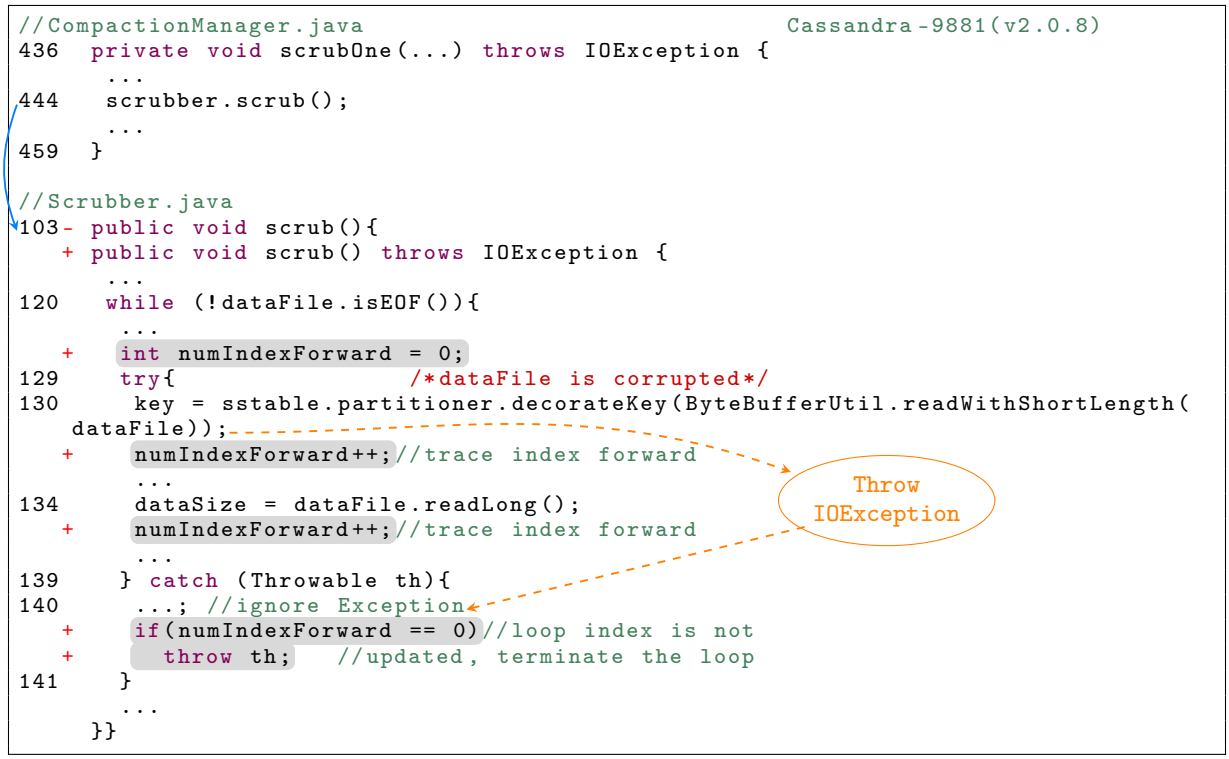

Figure 4.8 Example of hang bug pattern #3 and its fixing strategy. Data corruption causes `readWithShortLength()` to throw exception at line #130-131, which makes the loop skip the index updating statement (i.e., zero-stride) at line #134. "⟶" represents the function call invocation, while "- - >" represents the control flow. "-" means deleted code and "+" means added code, representing the patch generated by HangFix.

proper exception/error handling changes the control flow which causes the index-forwarding/reversing operations to be skipped, leading to an infinite loop. Note that, the bugs in this category are different from the bugs with the unexpected return values (pattern #1) in that this bug pattern skips the index updating operations while the pattern #1 bugs update the loop index incorrectly. Thus, their patching strategies are different.

Figure 4.8 shows the Cassandra-9881 bug caused by an improper exception handling. When `dataFile` is corrupted, an `IOException` is raised by the `readWithShortLength()` function at line #130. The `IOException` gives up the correct execution of `readWithShortLength()` and skips the `readLong()` function at line #134. The above read functions are both loop index-forwarding operations. This `IOException` is then simply ignored at line #140. Without moving the loop index forwards at line #130 and #134, the `scrub()` function keeps reading from the same place, spinning forever.

Figure 4.9 shows the Hive-18217 bug caused by an improper error handling. When the `Text from` is corrupted at line #224, the extracted `ByteBuffer fromBytes` at line #228 contains trailing bytes. Those trailing bytes make the `extraBytesToRead` variable be negative at line #618, which in turn makes the `bytesToPoint()` function skip executing the loop index-forwarding operation

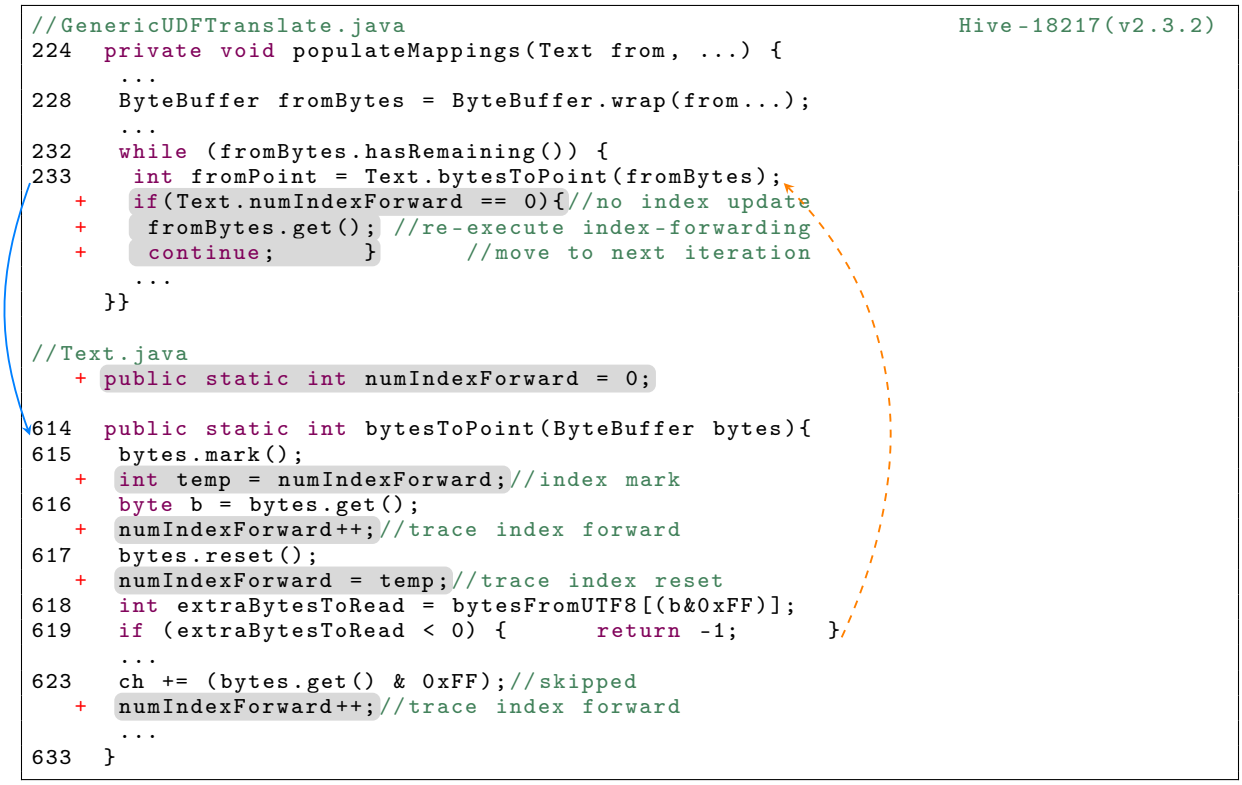

Figure 4.9 Example of hang bug pattern #3 and its fixing strategy. When `fromByte` is corrupted, the `extraBytesToRead` variable becomes negative, which makes the `bytesToPoint()` function return error code -1 at line #619 and skip the index-forwording statement (i.e., zero-stride) at line #623. "→" represents the function call invocation, while "- ->" represents the control flow. "+" means added code, representing the patch generated by HangFix.

`ByteBuffer.get()` at line #623, and return -1 at line #619. Without moving the loop index forwards, the `populateMappings()` function keeps processing the same trailing bytes and hangs.

Patching strategy #3: Tracing the execution of index update operations and re-executing them or breaking the loop. HangFix increases or decreases a counter value by one after each index-forwarding or index-reversing operation to trace their executions, and checks the counter value to decide whether a bug happens. If the loop index is not updated (e.g., index-forwarding/reversing APIs are not invoked, or index-resetting APIs are invoked) in the exception or error handling control flow, HangFix either 1) re-executes the loop index updating operations in each loop iteration or 2) terminates the loop by throwing a checkable and acceptable exception. Note that, our *function profile* contains the commonly used Java library function type information, i.e., index-forwarding API, index-reversing API, index-resetting API, index-marking API, which we can inquire during analysis.

To illustrate how HangFix fixes hang bugs which are caused by an improper exception handling, we use the Cassandra-9881 bug in Figure 4.8 as an example. HangFix traverses the CFG of the hang function `scrub()`, inserts a counter variable `numIndexForward` before the `try` block (line #129), extracts all the invocation statements from the `try` block, inquires the *function pro-*

file, and finds that the operations at line #130 and #134 move the loop index forwards. HangFix identifies `dataFile.readLong()` at line #134 as index-forwarding, because `dataFile` is a `RandomAccessReader` instance which implements the `DataInput` interface, and HangFix's *function profile* contains all the index-forwarding methods in the `DataInput` interface. HangFix identifies `readWithShortLength(dataFile)` at line #130 as index-forwarding, because HangFix detects its callee functions `readUnsignedShort()` and `readBytes()`[1] are index-forwarding methods in the `DataInput` interface. After line #130 and line #134, HangFix increases the counter by one, i.e., `numIndexForward++`. In the `catch` block, HangFix inserts an `if` branch with the condition of `numIndexForward == 0` and a `throw` exception statement to disrupt the loop. This is doable because HangFix adds the `throws IOException` clause in the method signature of `scrub()` and relies on Cassandra's existing exception handling mechanisms to repair failures by propagating the exception backwards to the caller function, `scrubOne()`. In the patched code, `scrub()` and `scrubOne()` share the same type of exceptions declared in their function signatures. Note that, HangFix does not re-execute the loop index updating operations, i.e., `readWithShortLength()` and `readLong()`, in the `catch` block due to two reasons. First, their execution affects many other variables' control and data flows, making the patch generation challenging and not generic. Second, re-executing them can still throw exceptions because of the corrupted `dataFile`, which makes the re-execution unnecessary and problematic.

To illustrate how HangFix fixes hang bugs which are caused by an improper error handling, we use the Hive-18218 bug in Figure 4.9 as an example. After traversing the CFG of the hang function `populateMappings()`, HangFix detects that the loop index updating operations are in its callee function `bytesToPoint()`. To trace and retrieve the execution status of those index updating functions from the callee, HangFix inserts a class field variable `numIndexForward` in the callee as a counter, increases the counter by one after the index-forwarding statements at line #616 and #623, and resets the counter to the previously saved value (line #615) after index-resetting statement at line #617. In the `populateMappings()` function, after the callee invocation statement at line #233, HangFix inserts an `if` branch with the condition of `Text.numIndexForward == 0` to check whether the loop index is moved forwards. Inside the `if` branch, HangFix inserts the re-invocation of the index-forwarding statement `fromBytes.get()`, where `fromBytes` is the argument passed into the callee function, followed by a `continue` statement. The added code handles the error returned from the callee by moving the loop index forwards to skip the corrupted byte in the current iteration. Thus, the `populateMappings()` function can finish processing the `fromBytes` and proceed the execution of the Hive system.

4.2.5 Fixing Hang Bugs Related to Blocking-Prone Operations

Hang bug pattern #4: Blocking-prone operations. This category of hang bugs involve inter-process communication, where component A endlessly waits for component B to finish before moving on. Component A either blocks itself waiting to be awakened by component B or keeps polling from

[1] To save space, we omit the two callee functions in Figure 4.8.

```
//YarnClientImpl.java                                          Yarn-1630(v2.2.0)
   + private String POLL_TIMEOUT_KEY ="poll.timeout";
   + private int DEFAULT_POLL_TIMEOUT = 5000;
   + private long timeout = conf.getInt(POLL_TIMEOUT_KEY, DEFAULT_POLL_TIMEOUT);

142  public ApplicationId submitApplication(...)
143        throws YarnException, IOException {
       ...
   +   long st = System.currentTimeMillis();
153    while(true) {
154     state = getAppReport(appId).getYarnAppState();
155     if (!state.equals(YarnAppState.NEW) && ...)
156      break;
       ...   //checking elapsed time
   +    long elapsed= System.currentTimeMillis() - st;
   +    if (timeout > 0 && elapsed >= timeout)
   +     throw new TimeoutException("Timed out. Breaking infinite polling!");
170    }
       ...
177  }
```

Figure 4.10 Example of hang bug pattern #4 and its fixing strategy. When the state of `appId` is asynchronized, the `submitApplication` function keeps polling its state but gets `NEW`, hanging in an infinite loop. "+" means added code, representing the patch generated by HangFix.

```
//TcpPeerServer.java                                           HDFS-10223(v2.7.0)
  + private Configuration conf = new Configuration();
  + private String PEERSEND_TIMEOUT_KEY = "peersend.timeout";
  + private int DEFAULT_PEERSEND_TIMEOUT = 60 * 1000;
  + private int timeout = conf.getInt(PEERSEND_TIMEOUT_KEY,
      DEFAULT_PEERSEND_TIMEOUT);

81  public static Peer peerFromSocketAndKey(...) throws IOException {
  +  peer.setReadTimeout(timeout);//provided API
90    peer = saslClient.peerSend(peer, ...);
       ...
98  }
```

Figure 4.11 Example of hang bug pattern #4 and its fixing strategy. Without setting timeout for network connection, the `TcpPeerServer` hangs on reading from an unresponsive DataNode. "+" means added code, representing the patch generated by HangFix.

component B until a variable state is changed or an operation finishes. Component A is where the hang function resides while component B can be another thread, another node, in underlying JNI code or even OS code which are out of our analysis scope. Since component B can be unresponsive while component A blocks its execution and keeps waiting, we call such communication to be blocking-prone operations. Our *function profile* contains all the Java library functions which are blocking-prone, for example, `SocketInputStream.read()`.

For example, the HBase-8389 bug shown in Figure 4.1 and Figure 4.2 falls in into this pattern. A similar case is the Yarn-1630 bug shown in Figure 4.10. It happens in an infinite loop, where the `submitApplication()` function keeps polling the state of an asynchronized application `appID` but always gets `YarnAppState.NEW`. This infinite polling loop is a blocking-prone operation. The HDFS-10223 bug, shown by Figure 4.11, happens when an HDFS client invokes the `peerSend()` operation to endlessly wait for a response from an unresponsive DataNode. The `peerSend()` function is

```
//ZlibCodec.java                                                    Hive-5235(v1.0.0)
 81 public void decompress(ByteBuffer in, ByteBuffer out) throws IOException {
 93  try {
 94-  int cnt = inflater.inflate(out.array(),
   +  int cnt = inflateWithTO(inflater, out.array(),
 95              out.arrayOffset() + out.position(), out.remaining());
     ...
 97  } catch (DataFormatException e) {
 98   throw new IOException("Bad compressed data",e);
 99  }
     ...
105 }

+private Configuration conf = new Configuration();
+private String INFLATE_TIMEOUT_KEY = "orc.zlibcodec.inflate.timeout";
+private long DEFAULT_INFLATE_TIMEOUT = 5000;
+private long timeout =conf.getLong(INFLATE_TIMEOUT_KEY, DEFAULT_INFLATE_TIMEOUT);

//a callable thread with timeout setting
+public int inflateWithTO(final Inflater inflater, final byte[] b, final int off,
    final int len) throws DataFormatException {
+ ExecutorService executor = Executors.newSingleThreadExecutor();
+ Callable<Integer> callable = new Callable<Integer>(){
+  @Override
+  public Integer call() throws DataFormatException {
+   return inflater.inflate(b, off, len);
+ }};
+ Future<Integer> future = executor.submit(callable);
+ int cnt = 0;
+ try {
+  cnt = future.get(timeout, TimeUnit.MILLISECONDS); //timeout setting
+ } catch (Exception e) {
+  future.cancel(true);
+  throw new DataFormatException("Endless blocking"); //acceptable exception
+ } finally { executor.shutdown(); }
+ return cnt;
+}
```

Figure 4.12 Example of hang bug pattern #4 and its fixing strategy. `Inflater.inflate()` is a blocking-pone function. When an ORC file is corrupted, conducting the `inflate()` operation on a corrupted file causes an infinite loop in the underlying JNI code. "⟶" represents the function call invocation. "-" means deleted code and "+" means added code, representing the patch generated by HangFix.

blocking-prone. The Hive-5235 bug shown in Figure 4.12 happens when `ZlibCodec` conducts the `inflate()` operation on a corrupted ORC file, causing an infinite loop in the underlying JNI code. The `decompress()` and `inflate()` functions are blocking-prone operations.

Patching strategy #4: Adding missing timeout over blocking-prone operations. For this type of bugs, HangFix inserts timeout settings considering three cases. First, when the bug happens in an infinite polling loop in the application code instead of underlying JNI code or OS code, HangFix calculates the elapsed time in each loop iteration and compares it with a pre-set timeout variable var_{new}. If the program has executed longer than var_{new}, HangFix breaks the loop by throwing a known exception. Second, when an application provides specific timeout setting APIs over the blocking-prone operations, HangFix inserts such API calls with timeout variable var_{new} before the blocking-prone operations. Third, HangFix quarantines the blocking-prone operation in a callable or runnable thread with timeout settings, e.g., `future.get(`var_{new}`)`, `thread.join(`var_{new}`)`.

The newly introduced timeout variable var_{new} is configurable and with the default value of v. HangFix extracts the default value by searching the system's configuration files using keywords, such as "timeout", "interval", "block", "poll". HangFix chooses the variable who matches the most keywords, and assigns its default value v to var_{new}. The rationale is that variables share similar names most likely have similar purposes, thus similar default value. In our future work, HangFix can use other timeout fixing/recommendation tools [He19] to assign a more accurate value to the var_{new} variable, which is out of the scope of this chapter.

The Yarn-1630 bug in Figure 4.10 can be fixed using this strategy. HangFix first introduces the `timeout` variable with default value of 5000 milliseconds. This default value is read from the `Job.DEFAULT_COMPLETION_POLL_INTERVAL` variable. Next, HangFix records the starting timestamp st by inserting the `System.currentTimeMillis()` function before the loop header (line #153). HangFix then retrieves the current timestamp $curr$ in each loop iteration, calculates the elapsed time $elapsed = curr - st$ and compares the elapsed time with the pre-set timeout variable `timeout` in the `if` branch before the loop tail (line #171). The branch condition is `timeout > 0` and `elapsed >= timeout` to make sure that timeout is set and the loop has used up the predetermined period of time. In the `if` branch, a `TimeoutException` is thrown with error message "Timed out. Breaking infinite polling". `TimeoutException` is the subtype of `IOException`, which is acceptable and matches our design. The HBase-8389 bug can be fixed in the same way shown by Figure 4.2. Thus, we do not rephrase its detailed patching steps.

To fix the HDFS-10223 bug in Figure 4.11, HangFix first introduces the `timeout` variable with default value of 60 seconds. This default value is read from the `HdfsServerConstants.READ_TIMEOUT` variable. Next, HangFix detects that the `peerSend()` function is a blocking-prone operation at line #90. This is because `peerSend()` has a seven-hop callee `SocketInputStream.read()`, and the `read()` function is blocking-prone, recorded in the *function profile*. To save space, we omit the call paths from `peerSend()` to `read()` in Figure 4.11. HangFix retrieves the timeout setting API `setReadTimeout()` in the `Peer` class. HangFix then inserts the `peer.setReadTimeout(timeout)` statement before `peerSend()` at line #90.

To fix the Hive-5235 bug in Figure 4.12, HangFix first introduces the `timeout` variable with default value of 5000 milliseconds. This default value is read from the `HiveConf.HIVE_SERVER2_LONG_POLLING_TIMEOUT` variable. Next, HangFix inquires the *function profile* and detects that the `decompress()` function contains a blocking-prone operation `Inflater.inflate()` at line #94. HangFix replaces it with a new function called `inflateWithTO()` using the `Executor-Callable-Future` format to quarantine the `inflate()` operation inside a callable thread. The new generated function uses both the `inflater` instance and all the arguments from `inflate()` as parameters. To seamlessly integrate with the `decompress()` function and the exception handling mechanisms in the Hive system, `inflateWithTO()` transforms the exception into `DataFormatException`. The timeout setting is in the `future.get()` function with the `timeout` variable and the `milliseconds` time unit.

Table 4.1 Hang bug benchmarks. Even though some bugs have the same description, they happen in different functions or classes.

Bug name	Version	Description
Cassandra-7330	v2.0.8	Skipping on a corrupted `InputStream` returns error code, affecting loop stride.
Cassandra-9881	v2.0.8	Reading a corrupted file throws exception and improper exception handling skips loop index-forwarding API.
Compress-87	v1.0	Reading on a truncated zip file returns error code, affecting loop stride.
Compress-451	v1.0	Misconfigured variable `bufferSize` indirectly affects loop index.
Hadoop-8614	v0.23.0	Skipping after EOF returns error code, affecting loop stride.
Hadoop-15088	v2.5.0	Skipping on a corrupted `InputStream` returns error code, affecting loop stride.
Hadoop-15415	v2.5.0	Misconfigured variable `buffSize` indirectly affects loop index.
Hadoop-15417	v2.5.0	Misconfigured variable `bufferSize` indirectly affects loop index.
Hadoop-15424	v2.5.0	Misconfigured variable `buff` causes loop stride be 0.
Hadoop-15425	v2.5.0	Misconfigured variable `sizeBuf` indirectly affects loop index.
Hadoop-15429	v2.5.0	Unsynchronized index is set and reset periodically, causing `DataInputByteBuffer` hangs.
HDFS-4882	v0.23.0	Corruption handling causes loop index update operation skipped.
HDFS-5438	v0.23.0	Incorrect block report processing causes corrupted replicas to be accepted during commit.
HDFS-10223	v2.7.0	`TcpPeerServer` endlessly waits for a response from an unresponsive DataNode.
HDFS-13513	v2.5.0	Misconfigured variable `BUFFER_SIZE` indirectly affects loop index.
HDFS-13514	v2.5.0	Misconfigured variable `BUFFER_SIZE` indirectly affects loop index.
HDFS-14481	v2.5.0	Misconfigured variable `BUFFER_SIZE` causes loop stride be 0.
HDFS-14501	v2.5.0	Misconfigured variable `BUFFER_SIZE` causes loop stride be 0.
HDFS-14540	v0.23.0	Block deletion failure causes an infinite polling.
Mapreduce-2185	v0.23.0	Improper error handling causes the loop index updating operation `rackToBlocks.put()` skipped.
Mapreduce-5066	v2.0.3	JobEndNotifier endlessly waits for a response from an unresponsive Hadoop job.
Mapreduce-6990	v0.23.0	Skipping on a corrupted `InputStream` returns error code, affecting loop stride.
Mapreduce-6991	v2.5.0	File creation failed due to exception and improper exception handling skips loop index-forwarding API.
Mapreduce-7088	v2.5.0	Misconfigured variable `bufferSize` causes loop stride be 0.
Mapreduce-7089	v2.5.0	Misconfigured variable `bufferSize` causes loop stride be 0.
Yarn-163	v0.23.0	Skipping on a corrupted `FileReader` returns error code, affecting loop stride.
Yarn-1630	v2.2.0	YarnClient endlessly polls the state of an asynchronized application.
Yarn-2905	v2.5.0	Skipping on a corrupted aggregated log file returns error code, affecting loop stride.
HBase-8389	v0.94.3	HBase endlessly sends lease recovery request to HDFS but HDFS fails on recovering corrupted blocks.
Hive-5235	v1.0.0	Uncompressing a corrupted ORC file blocks the Hive task.
Hive-13397	v1.0.0	Reading on a corrupted ORC file returns error code, affecting loop stride.
Hive-18142	v1.0.0	Reading on a corrupted ORC file returns error code, affecting loop stride.
Hive-18216	v2.3.2	Corrupted `ByteBuffer` makes `bytesToPoint` function returns eror code and skips loop index-forwarding API.
Hive-18217	v2.3.2	Corrupted `ByteBuffer` makes `bytesToPoint` function returns eror code and skips loop index-forwarding API.
Hive-18219	v2.3.2	Skipping on a corrupted `InputStream` returns error code, affecting loop stride.
Hive-19391	v1.0.0	RowContainer endlessly retries to create a file but failed.
Hive-19392	v1.0.0	Unsynchronized index is set and reset periodically, causing `DataInputByteBuffer` hangs.
Hive-19395	v1.0.0	Misconfigured variable `bufferSize` causes loop stride be 0.
Hive-19406	v2.3.2	HiveKVResultCache endlessly retries to create a file but failed.
Kafka-6271	v0.10.0	Skipping on a corrupted file returns error code, affeting loop stride.
Lucene-772	v2.1.0	Index corruption causes Lucene stuck on uncompression task.
Lucene-8294	v2.1.0	Misconfigured variable `bufferSize` causes loop stride be 0.

4.3 Evaluation

In this section, we present our experimental evaluations on HangFix. We first describe our evaluation methodology and then discuss our evaluation results in detail.

4.3.1 Evaluation Methodology

HangFix is written in Java, implemented on top of Soot compiler [Soo], using `BodyTransformer` and `SceneTransformer` to conduct intra- and inter-procedural analysis, and `ForwardFlowAnalysis` to conduct data flow dependency analysis.

Benchmarks: Our experimental evaluation covers 42 real-world hang bugs from 10 cloud server systems, shown in Table 4.1. The 10 systems include Cassandra key-value store, Compress I/O compression library, Hadoop common library, Hadoop Mapreduce big-data processing framework, Hadoop HDFS file system, Hadoop Yarn resource management service, HBase database management system, Hive data warehouse, Kafka streaming platform, and Lucene text searching engine.

Patching evaluation: We manually studied each bug report and wrote testcases to reproduce all 42 hang bugs. We ran our testing code to trigger the bug and made sure the software hang happens. We then generated the patch using HangFix on the buggy application's bytecode. We evaluate whether HangFix can fix a bug by re-running our testing code on the patched application bytecode. We evaluate whether the patch affects the program's normal execution flows by running the existing testcases in the corresponding application package. If the patched program executes successfully using both our bug-triggering code and application's existing testcases without hanging or crashing (e.g, throwing `RuntimeException` or `AssertionError`), we consider the patch is valid. We also compare HangFix with manual patches presented in bug reports in JIRA [Jir] to further validate the effectiveness of HangFix.

Setup: All the experiments were conducted in our lab machine with an Intel® i7-4790 Octa-core 3.6GHz CPU, 16GB memory, running 64-bit Ubuntu v16.04 with kernel v4.4.0.

4.3.2 Result Analysis

Correctness: As shown in Table 4.2, HangFix successfully fixes 40 out of 42 hang bugs in our benchmarks, including 13 bugs using patching strategy #1, 13 bugs using patching strategy #2, five bugs using patching strategy #3, and nine bugs using patching strategy #4. We manually tested the patched application code by re-running our bug triggering code and applications' existing testcases. Our experiments show that the patched program can execute successfully without hanging or crashing even when we trigger the bug, and execute correctly with no delay using the existing testcases. In contrast, the manual patch can only fix 14 out of the 42 bugs.

To further evaluate the patches generated by HangFix, we compare them with the manual patches for the 14 fixed bugs by both approaches. We find that HangFix's patches are similar as the manual ones except for the Compress-451 bug. Its manual patch throws runtime exceptions after identifying the `buffersize` variable is misconfigured to be non-positive, while HangFix breaks the

Table 4.2 The comparision of HangFix and manual fixing. "-" means the bug is unresolved. "*" means the developer closes the report without fixing the bug or the bug still happens after the deverlop closes the report.

Bug name	Manual		HangFix		
	Fixed?	Time (day)	Fixed?	Time (sec)	Patching strategy
Cassandra-7330	✓	4	✓	1.2±0.2	#1
Compress-87	✓	6	✓	1.1±0.1	#1
Hadoop-8614	✓	37	✓	0.7±0.1	#1
Hadoop-15088	×	-	✓	1.0±0.1	#1
Hadoop-15424	×	-	✓	0.9±0.1	#1
Hadoop-15425	×	-	✓	1.1±0.1	#1
Mapreduce-6990	×	-	✓	0.8±0.1	#1
Yarn-163	×	-*	✓	0.9±0.0	#1
Yarn-2905	✓	6	✓	0.8±0.0	#1
Hive-13397	✓	21	✓	0.8±0.1	#1
Hive-18142	×	-	✓	0.9±0.1	#1
Hive-18219	×	-	✓	1.0±0.1	#1
Kafka-6271	×	-	✓	0.9±0.0	#1
Compress-451	✓	1	✓	0.8±0.1	#2
Hadoop-15415	×	-	✓	0.9±0.1	#2
Hadoop-15417	×	-*	✓	22±1.0	#2
Hadoop-15429	×	-	✓	0.8±0.1	#2
HDFS-13513	×	-	✓	0.9±0.1	#2
HDFS-13514	×	-	✓	1.1±0.1	#2
HDFS-14481	×	-	✓	0.7±0.0	#2
HDFS-14501	×	-	✓	0.8±0.1	#2
Mapreduce-7088	×	-	✓	1.0±0.1	#2
Mapreduce-7089	×	-	✓	0.8±0.0	#2
Hive-19392	×	-	✓	0.9±0.1	#2
Hive-19395	×	-	✓	1.0±0.0	#2
Lucene-8294	✓	196	✓	1.0±0.1	#2
Cassandra-9881	×	-	✓	0.9±0.1	#3
HDFS-4882	✓	537	×	-	#3
Mapreduce-2185	✓	206	✓	1.3±0.2	#3
Mapreduce-6991	×	-	✓	1.2±0.1	#3
Hive-18216	×	-	✓	1.2±0.2	#3
Hive-18217	×	-	✓	1.1±0.2	#3
HDFS-10223	✓	2	✓	1.0±0.1	#4
HDFS-5438	✓	17	×	-	#4
HDFS-14540	×	-	✓	1.1±0.1	#4
Mapreduce-5066	✓	37	✓	0.9±0.1	#4
Yarn-1630	✓	5	✓	1.1±0.1	#4
HBase-8389	✓	24	✓	0.9±0.0	#4
Hive-5235	×	-	✓	0.8±0.1	#4
Hive-19391	×	-	✓	1.2±0.2	#4
Hive-19406	×	-	✓	0.8±0.1	#4
Lucene-772	×	-	✓	0.7±0.0	#4

loop and proceeds the program's execution. We applied the manual patch on the buggy Compress program, ran our bug-triggering code, and found that the program with the manual patch crashed after we triggered the bug.

Fixing time: HangFix fixes the 40 bugs in seconds, much faster than manual patch. Among the 14 manually fixed bugs, six bugs are fixed within one week, three are fixed within one month, two are fixed within two months, two are fixed within seven months, and one is fixed one and a half years later. As for the remaining unfixed 28 bugs, they are reported years ago but are still unresolved. Note that, the Hive-13397 report is resolved in less than a day because the fixing is the same as the

```
//LeaseManager.java                                          HDFS-4882(v0.23.0)
369  public void run() {
370   for(; fsnamesystem.isRunning(); ) {
       ...
374    checkLeases();
       ...
388  }}

393 private synchronized void checkLeases() {
     ...
395  for(; sortedLeases.size() > 0; ) {
396   final Lease oldest = sortedLeases.first();
     ...
   +  int numIndexForward = 0;
412  if(fsnamesystem.internalReleaseLease(p, ...)) { //p is a file's lease path
413   LOG.info("...");
414   removing.add(p);//remove p from sortedLeases
   +   numIndexForward++;
416  } else {
417   LOG.info("...block recovery for file " + p);
418  }
   +  if(numIndexForward == 0) removing.add(p);
     ...
429 }}
```

Figure 4.13 Example of a pattern #3 hang bug which cannot be fixed by HangFix. A corrupted file `f` associated with the lease path `p` makes the the lease recovery failed for `f` at line #412. When it happens, `p` is not removed from `sortedLeases` (skip updating loop index), LeaseManager keeps recovering lease for the file f endlessly. "⟶" represents the function call invocation, while "- ->" represents the control flow. "+" means added code, representing the patch generated by HangFix.

Hive-13255 bug, we use the manual fixing time 21 days of Hive-13255 as the manual fixing time of Hive-13397.

We should note that, it might be unfair to directly compare HangFix's patch generation time with the bug's manual resolve time because HangFix is an automatic fixing tool while manual patches involve human effects and inevitably introduce latency. However, we believe that HangFix can help developers to efficiently fix hang bugs in massive cloud systems.

HangFix fixes most of the 40 hang bugs in about one second, except for the Hadoop-15417 bug, due to different implementations. HangFix supports inter-procedural analysis using either iterative `BodyTransformer` or `SceneTransformer` from Soot. `BodyTransformer` is faster but can fail sometimes while `SceneTransformer` is relatively slower but always succeeds. HangFix first generates the patch using iterative `BodyTransformer` and evaluates the patch. If the patched code cannot fix the bug, HangFix then uses the `SceneTransformer` approach.

4.3.3 Negative Case Study

There are two bugs (i.e., HDFS-4882 and HDFS-5438) HangFix cannot fix, because fixing them requires not only moving forward the hanging function but also recovering corrupted data.

In the HDFS-4882 bug, shown by Figure 4.13, HangFix inserts the loop index updating operations for re-execution in the patch after line #418. However, this patching strategy can only move forward the hanging function `checkLeases()`. Since HangFix does not have the knowledge about how to re-

store or repair the corrupted data, it cannot insert the `fsnamesystem.getEditLog().logSync()` statement after the `checkLeases()` function at line #374 to invoke the HDFS synchronization mechanism to restore the corrupted block. Without adding the `logSync()` operation, HDFS repeats on processing the corrupted block endlessly in a for loop at line #370-388.

In the HDFS-5438 bug, HangFix successfully inserts the timeout settings to break an infinite polling loop. However, this patching strategy can only move forward the hanging function `completeFile()`, it cannot restore the corrupted blocks in the pipeline recovery. As a result, corrupted replicas are accepted causing a further hang problem in the `DFSInputStream.fetchBlockByteRange()` function which does not release a lock called `CountDownLatch`.

HangFix cannot fix the above two hang bugs because it currently does not have the capability to recover a corrupted data, which is beyond the scope of this chapter and is part of our future work.

4.4 Summary

In this chapter, we have presented HangFix, an application-agnostic hang bug fixing tool for automatically recovering a cloud service outage caused by unexpected data processing or inter-process communication failures. HangFix identifies different hang bug patterns and produces corresponding software patches based on automatically generated patching strategies using intra-/inter-procedural analysis and data-dependency analysis. We have implemented a prototype of HangFix and evaluated it on 42 real-world software hang bugs in 10 commonly used cloud server systems. HangFix successfully fixes 40 out of 42 hang bugs within seconds.

CHAPTER

5

RELATED WORK

5.1 Static rule-based performance bug detection

Much work has been done to develop static bug detection tools. Each work uses pre-defined heuristics/rules to specifically target certain types of performance bugs. Jin et al. [Jin12b] employ rule-based methods to detect performance bugs that violate efficiency rules that have been violated before. Chen et al. [Che14] detect database related performance anti-patterns, like fetching excessive data from database and issuing queries that could have been aggregated. There are also tools that detect loop break conditions [Son14], inefficient nested loops [Nis13b] and workload-dependent loops [Xia13]. These bug-detection tools are only suitable for bug detections, not for diagnosing specific performance bugs occurred in production environments. Once applied for performance diagnosis, they will suffer the false positive and false negative problems discussed in §2.1.

Previous work Carburizer [Kad09] statically analyzes device driver code and identifies infinite driver-polling problems. That is, a driver may wait for a device to enter a given state by polling a device register. Once the register data is corrupted, a buggy driver may be stuck forever. DScope and Carburizer both statically analyze loops and loop-exit conditions. However, they face different design challenges due to the different types of bugs they target. DScope targets cloud systems written in Java, instead of low-level device drivers, and hence needs to handle a much broader set of I/O functions and I/O related data (e.g., not only data retrieved by I/O operations but also status returned by I/O operations), and more complicated control flows caused by Java exceptions. Carburizer false-positive pruning only involves identifying loop time-outs. However, DScope has to conduct sophisticated loop stride and bound analysis in its false-positive pruning. Finally, as indicated in §3.3, the type of corruption-hang bugs identified by DScope in cloud systems go much

beyond simple I/O-state infinite polling problems, where the device register content often directly updates the loop stride.

5.2 On-site performance bug diagnosis

Dynamic analysis techniques have been used to identify and fix performance bugs that are triggered in production environments. X-ray [Att12] uses symbolic execution to automatically identify and suggest fixes to performance bugs caused by configuration or input-based problems. Strider [Wan03] 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, the root cause comes from other sources (e.g., unexpected component interactions, unexpected return value, incorrectly handled exceptions).

PerfCompass [Dea15b] focuses on differentiating external faults (e.g., interference from co-located applications) from internal faults (e.g., software bugs) for system performance anomalies. IntroPerf [Kim14] automatically infers the latency of user-level and kernel-level function calls based on OS tracers. StackMine [Han12] automatically identifies certain call stack patterns that are correlated with performance problems of event handlers. All these diagnosis tools are very useful in practice, but have different focus from our work. They do not aim to identify root cause related functions of performance problems.

Many techniques have been proposed to diagnose performance problems in distributed systems. For example, Aguilera et al. [Agu03] identify the performance bottleneck nodes by conducting causal path analysis on the message-level traces (e.g., RPC message). Kasick et al. [Kas10] identify the faulty components (e.g., storage or network) by statistically debugging the OS-level metrics (e.g., I/O requests rate, packet reception rate). Xu et al. [Xu09] and CloudSeer [Yu16] detect the faulty nodes by mining the workflows from the console logs. Those tools focus on identifying the faulty components, nodes or interactions that lead to performance problems, which are different from our work (i.e., identifying root cause related functions).

5.3 Hybrid bug diagnosis

Hybrid techniques have been used to fix concurrency bugs. For example, AFix [Jin11] and CFix [Jin12a] statically analyze blocking operations (e.g., lock-acquisitions, condition-wait and thread join operations) as potential failure points to construct a concurrency bug patch and perform dynamic runtime testing to evaluate the effectiveness of the patch. Previous work also uses static analysis and dynamic instrumentation for statistical debugging. For example, HOLMES [Chi09] statically identifies potential buggy code regions using given failure points and stack trace, and instruments the program to profile those regions. It then dynamically analyzes the collected profiles from subsequent runs of the program to identify the root cause. Work has also been done to perform replay debugging by combining static analysis with symbolic execution. For example, ESD [Zam10] statically identifies

candidate paths that can reach a failure point and symbolically executes the program to synthesize the failure-triggering input. In contrast, our work does not require failure points, error statements, or application instrumentation, which makes it more practical for diagnosing performance bugs in production cloud environments.

5.4 Data corruption study and detection

Previous work has been extensively studied the data corruption problems in storage systems. Hwang et al. [Hwa12] and Schroeder et al. [Sch09] studied the data corruptions in memory devices. They found that DRAM failures occur more frequently than expected. Bairavasundaram et al.[Bai08a; Bai08b] and Oleksenko et al. [Ole16] detected the data pointer corruptions on disks. They showed that disk failures are prevalent for data corruptions. Previous works have also been done to detect data corruptions in file systems. ZFS [Bon07] detected file system corruption caused by storage hardware, e.g., latent sector errors. Fryer et al. [Fry14; Sun14] implemented runtime data corruption detectors for the Ext3 and Btrfs file systems.

The above work provides motivations for us to study data corruption induced performance problems. Our work focuses on detecting data corruption hang bugs in software-level rather than detecting the data corruption itself (hardware-level).

5.5 Automatic bug fixing:

Work has been done for automatic bug fixes. AFix [Jin11] fixes atomicity violation bugs by quarantining critical sections using locks. CFix [Jin12a] fixes concurrency bugs by enforcing the order relationship of synchronization operations to prevent the buggy interleaving. ClearView [Per09] fixes invariant violation bugs by enforcing the buggy invariant to be true after changing its control state and control flow. TFix [He19] proposes a drill-down bug analysis approach to identify timeout bug's root cause and suggest correct timeout values. DFix [Li19] adopts the rollback and fast-forward strategy to fix distributed timing bugs. Tufano et al. [Tuf18] applies an encoder/decoder model to mine the existing patches and automatically generate new patches. Genprog [Le 12] is a search-based genetic programming approach for automated program repairs. SemFix [Ngu13b] is a semantic-based program repair tool, which derives repair constraints from a set of tests and solves the repair constraints to generate a valid repair. Assure [Sid09] fixes runtime faults by restoring program execution to a rescue point where error-handling is performed to recover the program execution. Ares [Gu16] recovers the program from runtime unexpected errors with the program's existing error-handlers. Ares synthesizes a number of error-handlers and selects the most promising one via virtual testing techniques. Compared to those existing bug fixing schemes, HangFix focuses on fixing software hang bugs caused by data processing or inter-process communication failures in the cloud system based on four application-agnostic hang patterns.

5.6 Fault injection

Previous work [Bar90; Gu03; Kou16] used fault injection techniques to analyze the failure behaviors (e.g., hang, crash) of both software and hardware systems. For example, HSFI [Kou16] injected faults in the source code. Fault injection is also widely used to check whether file systems can handle certain type of data corruptions [Gan17; Fia12; Pra05; Zha10]. For instance, Bairavasundaram et al. [Bai06] used context aware fault injections to find disk errors in virtual memory systems. Zhang et al. [Zha10] conducted a comprehensive reliability case study of local file systems to analyze both on-disk and in-memory data integrity in Sun's ZFS. Their results show that file systems are robust to disk corruption but less resilient to memory corruptions. Cords [Gan17] exposed data losses, block corruptions, and unavailability problems commonly exist in distributed file systems. Cords also indicated that modern distributed file systems are not equipped to effectively use redundancy across replicas to recover from local file system faults. In contrast, our work focuses on detecting potential data corruption hang bugs before they are triggered by the data corruption faults. We only rely on static code analysis, which can be easily applied to different cloud server systems. We believe our work is complementary to the fault injection based approaches which can be used to validate our candidate bugs and further reduce false positives.

5.7 Functional bug detection

Apart from performance bugs, recent works have also been done to detect functional bugs. pbSE [Xia17] conducted concolic execution to detect functional bugs and generate test cases for those bugs. Kollenda et al. [Kol17] detected the crash bugs by identifying the crash-resistant primitives via system calls on Linux, Windows API functions, and exception handlers. In contrast, our work focuses on detecting performance bugs, which requires the bug detection system to focus on different aspects of the program such as loop exit checking.

5.8 Software testing

DeepXplore [Pei17] is a whitebox framework to test deep learning systems. DeepXplore takes unlabeled test inputs as seeds in DNN systems. It uses gradient ascent to modify the input to maximize chance of finding rare corner cases. Fex [Ole17] is a software system evaluator, which collects a set of reused scripts to develop a matured evaluation framework. Fex addressed the limitation of rigid, simplistic and inconsistent in large system testing. Elia et al. [Eli15] designed an interoperability certification model, which facilitates testing interoperability among different web applications. Our work is complementary to those software testing tools. Our tool can identify potential buggy functions with infinite loops, which can guide the test case generation to further test our detection results.

CHAPTER

6

CONCLUSIONS AND FUTURE WORK

This report focuses on developing three key techniques for detecting, diagnosing and fixing performance bugs: diagnosing real-world performance bugs using both static and dynamic analysis, detecting data corruption hang bugs using pattern-driven static analysis, and automatically fixing software hang bugs caused by data-processing or inter-process communication failures using both static patch generation and dynamic patch testing. These three motivate and complement each other. They all have been proven to be efficient and effective for expediting the detection, diagnosis and fix of performance bugs in cloud server systems.

The organization of this chapter is as follows. First, we summarize our main contributions. We then briefly describe possible future research directions.

6.1 Contributions

This report makes the following specific contributions in an attempt to design and implement novel techniques to handle performance bugs:

- We have presented Hytrace, a hybrid approach to diagnosing real-world performance bugs in production cloud systems. Hytrace combines rule based static analysis and runtime inference techniques to achieve higher accuracy than pure-static or pure-dynamic approaches. Hytrace does not require any application source code or instrumentation, which makes it practical for production cloud environments. We have implemented a prototype of Hytrace and tested it over 133 real performance bugs discovered in different commonly used server applications. Our results show that Hytrace can greatly improve coverage and precision comparing with

existing state-of-the-art techniques. Hytrace is light-weight, which imposes less than 3% CPU overhead to the testing cloud environments.

- We have presented DScope, a new data corruption hang bug detection tool for cloud server systems. DScope combines candidate bug discovery and false positive pattern filtering to detect software hang bugs that are related to data corruptions. DScope is fully automatic without requiring any user input or pre-defined rules. We have implemented a prototype of DScope and evaluated it over 9 commonly used cloud server systems. DScope successfully detects 42 true corruption hang bugs (29 of them are new bugs) while existing bug detection tools can only detect very few of them (2 by Findbugs and 1 by Infer).

- We have presented HangFix, an application-agnostic hang bug fixing tool for automatically recovering a cloud service outage caused by unexpected data processing or inter-process communication failures. HangFix identifies different hang bug patterns and produces corresponding software patches based on automatically generated patching strategies using intra-/inter-procedural analysis and data-dependency analysis. We have implemented a prototype of HangFix and evaluated it on 42 real-world software hang bugs in 10 commonly used cloud server systems. HangFix successfully fixes 40 out of 42 hang bugs within seconds.

6.2 Future Work

In this report, we have shown that our performance bug detection, diagnosis and fix framework can effectively identify the root-cause functions, detect data corruption infinite loops, and automatically correct software hang bugs caused by data processing and inter-process communication failures. In future, we plan to extend our approach in the following directions.

- **Supporting hybrid diagnosis on distributed performance bugs.** Our hybrid performance bug diagnosis tool, Hytrace focuses on single node performance bugs. For distributed performance bugs, Hytrace's diagnosis schemes are still preliminary. It generates a consolidated buggy function list by taking the intersection among all the buggy function lists produced by different faulty nodes. However, distributed system bugs can manifest as a chain of abnormal functions over multiple dependent nodes. Hytrace currently does not consider such causal relationships between distributed components. Previous work (e.g., FChain [Ngu13a], PCatch [Li18]) has developed distributed bug diagnosis tools based on distributed system causal analysis. Hytrace can integrate with those tools to achieve more precise distributed system performance bug diagnosis.

- **Integrating inter-procedural analysis on DScope.** Our static data corruption hang bug detection tool, DScope focuses on intra-procedural analysis. For inter-procedural data corruption induced hang bugs (e.g., HDFS-5438), DScope cannot detect them and inevitably introduces false negatives. DScope can add inter-procedural analysis in the candidate bug discovery

module to to achieve higher detection coverage. Currently, DScope also falsely reported 37 cases, in which, the loop index updating Java APIs and the bound checking Java APIs are located in different application functions. These APIs are indirectly invoked in the application functions which are invoked in the loop paths. To further reduce false positives, we plan to integrate inter-procedural analysis in the false positive pattern filtering module to to achieve higher detection accuracy.

- **Strengthening the fixing strategies on HangFix.** Our software hang bug fixing tool, HangFix fixes hang bugs by moving forward the hanging function using different strategies. For hang bugs (e.g., HDFS-4882, HDFS-5438) whose fix requires both moving forward the hanging function and recovering corrupted data, HangFix cannot correct the bugs. Recovering the corrupted data requires system specific information, which HangFix, as an application-agnostic tool, is lack of. To strengthen the fixing strategies, we plan to customize HangFix on popular cloud systems and propose specific fix to correct those bugs.

Performance bugs are a serious issue in the cloud today and the initial work done here by no means makes this a solved problem. A broader challenge researchers need to address is how to perform research at scale and in practice. That is, what types of complex performance bugs the large-scale production systems have and how to integrate the existing state-of-the-art approaches in production. Our work presented in this dissertation attempts to deliver an efficient and effective framework to make performance bug detection, diagnosis and fix easy. However, I believe there are a lot more opportunities to do better.

BIBLIOGRAPHY

[Agr94] Agrawal, R. & Srikant, R. "Fast algorithms for mining association rules". *VLDB*. 1994.

[Agu03] Aguilera, M. K. et al. "Performance Debugging for Distributed Systems of Black Boxes". *SOSP*. 2003.

[Bug] *Apache Bugzilla*. https://bz.apache.org/bugzilla/.

[Cas] *Apache Cassandra*. http://cassandra.apache.org/. 2019.

[Had] *Apache Hadoop*. http://hadoop.apache.org/. 2019.

[Hba] *Apache HBase*. http://hbase.apache.org. 2014.

[Jir] *Apache JIRA*. https://issues.apache.org/jira. 2019.

[Kaf] *Apache Kafka*. http://kafka.apache.org/. 2019.

[App] *App Engine*. https://cloud.google.com/appengine/.

[Aru13] Arulraj, J. et al. "Production-run software failure diagnosis via hardware performance counters". *ASPLOS*. 2013.

[Att12] Attariyan, M. et al. "X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software". *OSDI*. 2012.

[Aws] *AWS outage knocks Amazon, Netflix, Tinder and IMDb in MEGA data collapse*. https://www.theregister.co.uk/2015/09/20/aws_database_outage/.

[Bai06] Bairavasundaram, L. N. et al. "Dependability Analysis of Virtual Memory Systems". *DSN*. 2006.

[Bai08a] Bairavasundaram, L. N. et al. "An Analysis of Data Corruption in the Storage Stack". *TOS* **4**.3 (2008), 8:1–8:28.

[Bai08b] Bairavasundaram, L. N. et al. "Analyzing the Effects of Disk-Pointer Corruption". *DSN*. 2008.

[Bar90] Barton, J. H. et al. "Fault Injection Experiments Using FIAT". *TC* **39**.4 (1990).

[Bha08] Bhatia, S. et al. "Lightweight, High-resolution Monitoring for Troubleshooting Production Systems". *OSDI*. 2008.

[Bon07] Bonwick, J. & Moore, B. *ZFS–The Last Word In File Systems*. https://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf. 2007.

[Bor11] Borisov, N. et al. "Dealing Proactively with Data Corruption: Challenges and Opportunities". *SMDB*. 2011.

[Car11] Carbin, M. et al. “Detecting and Escaping Infinite Loops with Jolt”. *Proceedings of the European Conference on Programming Languages*. ECOOP. 2011.

[Che14] Chen, T.-H. et al. “Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping”. *ICSE*. 2014.

[Chi09] Chilimbi, T. M. et al. “HOLMES: Effective Statistical Debugging via Efficient Path Profiling”. *ICSE*. 2009.

[Ste] *Cisco Stealthwatch*. https://www.cisco.com/c/en/us/products/security/stealthwatch/index.html.

[Cla] *Clang*. https://clang.llvm.org/. 2018.

[Cot09] Cotroneo, D. et al. “Assessment and Improvement of Hang Detection in the Linux Operating System”. *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*. SRDS. 2009.

[Dai17] Dai, T. et al. “Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures”. *SoCC*. 2017.

[Dai18a] Dai, T. et al. “DScope: Detecting Real-World Data Corruption Hang Bugs in Cloud Server Systems”. *Proceedings of the ACM Symposium on Cloud Computing*. SoCC. 2018.

[Dai18b] Dai, T. et al. “Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures”. *IEEE Transactions on Parallel and Distributed Systems* (2018).

[Dai18c] Dai, T. et al. “Understanding Real World Timeout Problems in Cloud Server Systems”. *Proceedings of IEEE International Conference on Cloud Engineering*. IC2E. 2018.

[Dea12] Dean, D. J. et al. “UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems”. *ICAC*. 2012.

[Dea14] Dean, D. J. et al. “PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures”. *SoCC*. 2014.

[Dea15a] Dean, D. J. et al. “Automatic Server Hang Bug Diagnosis: Feasible Reality or Pipe Dream?” *Proceedings of IEEE International Conference on Autonomic Computing*. ICAC. 2015.

[Dea15b] Dean, D. J. et al. “PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds”. *TPDS*. 2015.

[Des06] Desnoyers, M. & Dagenais, M. R. “The lttng tracer: A low impact performance and behavior monitor for gnu/linux”. *Linux Symposium*. 2006.

[Ec2] *EC2*. https://aws.amazon.com/ec2/.

[Eli15] Elia, I. A. et al. “Test-Based Interoperability Certification for Web Services”. *DSN*. 2015.

[Emb] *Embarrassment*. https://dom.as/2009/06/26/embarrassment/.

[Fbi] *Facebook Infer*. http://fbinfer.com/. 2019.

[Fia12] Fiala, D. et al. “Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing”. *SC*. 2012.

[Fin] *Findbugs*. http://findbugs.sourceforge.net/. 2019.

[Fry14] Fryer, D. et al. “Checking the Integrity of Transactional Mechanisms”. *TOS* **10**.4 (2014), 17:1–17:23.

[Gan17] Ganesan, A. et al. “Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions”. *FAST*. 2017.

[Gu16] Gu, T. et al. “Automatic Runtime Recovery via Error Handler Synthesis”. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE. 2016.

[Gu03] Gu, W. et al. “Characterization of Linux Kernel Behavior Under Errors”. *DSN*. 2003.

[Gun14] Gunawi, H. S. et al. “What Bugs Live in the Cloud?: A Study of 3000+ Issues in Cloud Systems”. *Proceedings of the ACM Symposium on Cloud Computing*. SoCC. 2014.

[Han12] Han, S. et al. “Performance Debugging in the Large via Mining Millions of Stack Traces”. *ICSE*. 2012.

[Hdf] *HDFS-4882*. https://issues.apache.org/jira/browse/HDFS-4882. 2013.

[He18] He, J. et al. “TScope: Automatic Timeout Bug Identification for Server Systems”. *Proceedings of IEEE International Conference on Autonomic Computing*. ICAC. 2018.

[He19] He, J. et al. “TFix: Automatic Timeout Bug Fixing in Production Server Systems”. *Proceedings of IEEE International Conference on Distributed Computing Systems*. ICDCS. 2019.

[Hua15] Huang, J. et al. “Understanding Issue Correlations: A Case Study of the Hadoop System”. *SoCC*. 2015.

[Hwa12] Hwang, A. A. et al. “Cosmic Rays Don’t Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design”. *ASPLOS*. 2012.

[Hyt] *Hytrace*. http://dance.csc.ncsu.edu/projects/sysMD/Hytrace.html.

[Ins] *Instagram*. https://www.instagram.com/.

[Dyn] *Irreversible Failures: Lessons from the DynamoDB Outage*. http://blog.scalyr.com/2015/09/irreversible-failures-lessons-from-the-dynamodb-outage/. 2015.

[Jia08] Jiang, W. et al. “Is Disk the Dominant Contributor for Storage Subsystem Failures? A Comprehensive Study of Failure Characteristics”. *FAST*. 2008.

[Jin11] Jin, G. et al. “Automated atomicity-violation fixing”. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2011.

[Jin12a] Jin, G. et al. “Automated concurrency-bug fixing”. *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. OSDI. 2012.

[Jin12b] Jin, G. et al. “Understanding and Detecting Real-World Performance Bugs”. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2012.

[Jin12c] Jin, W. & Orso, A. “BugRedux: Reproducing Field Failures for In-house Debugging”. *ICSE*. 2012.

[Kad09] Kadav, A. et al. “Tolerating Hardware Device Failures in Software”. *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. SOSP. 2009.

[Kal17] Kaldor, J. et al. “Canopy: An End-to-End Performance Tracing And Analysis System”. *SOSP*. 2017.

[Kas10] Kasick, M. P. et al. “Black-box Problem Diagnosis in Parallel File Systems”. *FAST*. 2010.

[Kau09] Kaufman, L. & Rousseeuw, P. J. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009.

[Kim14] Kim, C. H. et al. “IntroPerf: Transparent Context-sensitive Multi-layer Performance Inference Using System Stack Traces”. *SIGMETRICS*. 2014.

[Kol17] Kollenda, B. et al. “Towards Automated Discovery of Crash-Resistant Primitives in Binary Executables”. *DSN*. 2017.

[Kou16] Kouwe, E. van der & Tanenbaum, A. S. “HSFI: Accurate Fault Injection Scalable to Large Code Bases”. *DSN*. 2016.

[Kut10] Kutare, M. et al. “Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers”. *ICAC*. 2010.

[Le 12] Le Goues, C. et al. “GenProg: A Generic Method for Automatic Software Repair”. *IEEE Trans. Software Eng.* **38**.1 (2012), pp. 54–72.

[Li19] Li, G. et al. “DFix: Automatically Fixing Timing Bugs in Distributed Systems”. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2019.

[Li18] Li, J. et al. “PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems”. *EuroSys*. 2018.

[Liu14] Liu, Y. et al. “Characterizing and Detecting Performance Bugs for Smartphone Applications”. *Proceedings of the 36th International Conference on Software Engineering*. ICSE. 2014.

[Llv] *LLVM*. http://llvm.org/.

[Mea09] Mearian, L. *Facebook temporarily loses more than 10% of photos in hard drive failure*. http://www.computerworld.com/s/article/9129263/Facebook_temporarily_loses_more_than_10_of_photos_in_hard_drive_failure. 2009.

[Mic] *MicroStrategy Analytics and Mobility*. https://www.microstrategy.com/.

[Mys14] Mysore, R. N. et al. "Gestalt: Fast, Unified Fault Localization for Networked Systems". *USENIX ATC*. 2014.

[Vcl] *NCSU Virtual Computing Lab*. http://vcl.ncsu.edu/.

[Ngu13a] Nguyen, H. et al. "FChain: Toward Black-box Online Fault Localization for Cloud Systems". *ICDCS*. 2013.

[Ngu14] Nguyen, H. et al. "Insight: In-situ Online Service Failure Path Inference in Production Computing Infrastructures". *USENIX ATC*. 2014.

[Ngu13b] Nguyen, H. D. T. et al. "SemFix: Program Repair via Semantic Analysis". *Proceedings of the 2013 International Conference on Software Engineering*. ICSE. 2013.

[Nis13a] Nistor, A. et al. "Discovering, Reporting, and Fixing Performance Bugs". *MSR*. 2013.

[Nis13b] Nistor, A. et al. "Toddler: Detecting Performance Problems via Similar Memory-Access Patterns". *ICSE*. 2013.

[Nis15] Nistor, A. et al. "Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes". *ICSE*. 2015.

[Ole16] Oleksenko, O. et al. "Efficient Fault Tolerance using Intel MPX and TSX". *DSN*. 2016.

[Ole17] Oleksenko, O. et al. "Fex: A Software Systems Evaluator". *Proceedings of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN. 2017.

[Oss] *OSSEC - Host-based Intrusion Detection System*. https://www.ossec.net/.

[Pat12] Patnaik, D. et al. "Efficient Episode Mining of Dynamic Event Streams." *ICDM*. 2012.

[Pei17] Pei, K. et al. "DeepXplore: Automated Whitebox Testing of Deep Learning Systems". *SOSP*. 2017.

[Per09] Perkins, J. H. et al. "Automatically patching errors in deployed software". *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP. 2009.

[Pra05] Prabhakaran, V. et al. "IRON File Systems". *SOSP*. 2005.

[Sah14] Saha, R. K. et al. "An Empirical Study of Long Lived Bugs". *CSMR-WCRE*. 2014.

[Sap] *SAP Business Intelligence Solutions*. https://www.sap.com/products/analytics/business-intelligence-bi.html.

[Sas] *SAS Business Analytics*. https://www.sas.com/en_us/solutions/business-analytics.html.

[Sch09] Schroeder, B. et al. “DRAM Errors in the Wild: A Large-Scale Field Study”. *SIGMETRICS*. 2009.

[She09] Shen, K. et al. “Reference-driven Performance Anomaly Identification”. *SIGMETRICS*. 2009.

[Sid09] Sidiroglou, S. et al. “ASSURE: Automatic Software Self-healing Using Rescue Points”. *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. 2009.

[Sno] *Snort - Network Intrusion Detection and Prevention System*. https://www.snort.org/.

[Son14] Song, L. & Lu, S. “Statistical Debugging for Real-World Performance Problems”. *OOPSLA*. 2014.

[Son17] Song, L. & Lu, S. “Performance Diagnosis for Inefficient Loops”. *ICSE*. 2017.

[Soo] *Soot: A Framework for Analyzing and Transforming Java and Android Applications*. https://sable.github.io/soot/. 2019.

[Sun14] Sun, K. et al. “Robust Consistency Checking for Modern Filesystems”. *RV*. 2014.

[Tan05] Tan, P. N. et al. *Introduction to Data Mining*. Addison Wesley, 2005.

[Tan12] Tan, Y. et al. “PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems”. *ICDCS*. 2012.

[Tuf18] Tufano, M. et al. “An empirical investigation into learning bug-fixing patches in the wild via neural machine translation”. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE. 2018.

[Twi] *Twitter*. https://twitter.com/.

[Ver96] Verbrugge, C. *Using Coffi*. http://www.sable.mcgill.ca/ clump/Coffi/Coffi.ps. 1996.

[Wan15] Wang, P. et al. “Understanding Real World Data Corruptions in Cloud Systems”. *Proceedings of IEEE International Conference on Cloud Engineering*. IC2E. 2015.

[Wan08] Wang, X. et al. “Hang Analysis: Fighting Responsiveness Bugs”. *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*. EuroSys. 2008.

[Wan03] Wang, Y. M. et al. “Strider: A black-box, state-based approach to change and configuration management and support”. *LISA*. 2003.

[Bac] *What Lessons can be Learned from BA’s Systems Outage?* http://www.extraordinarymanagedservices.com/news/what-lessons-can-be-learned-from-bas-systems-outage/. 2017.

[Xia17] Xiao, Q. et al. “pbSE: Phase-Based Symbolic Execution”. *DSN*. 2017.

[Xia13] Xiao, X. et al. “Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks”. *ISSTA*. 2013.

[Xu09] Xu, W. et al. “Detecting Large-scale System Problems by Mining Console Logs”. *SOSP*. 2009.

[Yu16] Yu, X. et al. “CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs”. *ASPLOS*. 2016.

[Zam10] Zamfir, C. & Candea, G. “Execution Synthesis: A Technique for Automated Software Debugging”. *EuroSys*. 2010.

[Zha10] Zhang, Y. et al. “End-to-End Data Integrity for File Systems: A ZFS Case Study”. *FAST*. 2010.