

ABSTRACT

DAI, TING. A Hybrid Approach to Cloud System Performance Bug Detection and Diagnosis. (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 data corruption hang bugs, we plan to present an auto-fix scheme to statically generate and dynamically evaluate patches for the bugs.

CHAPTER

1

INTRODUCTION

1.1 Motivation

Cloud computing infrastructures [7, 28] 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 [30]. Well tested cloud products such as Hadoop and Cassandra are also affected by performance bugs [21, 22, 46]. When performance bugs happen, they severely impact cloud serve performance and quality, prolonging serve response time, reducing system throughput, wasting system resources, and leading to poor user experience[62].

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[17, 46, 48, 63, 73, 84]. These bug-detection tools are suitable for detecting specific performance bugs. Once applied for generic performance diagnosis or other type performance bug detection, they will suffer 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 [2, 9, 26, 38, 50, 52, 81, 85, 86]. 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 and diagnosing performance bugs in the software level (e.g., code analysis).

Hybrid techniques have been used to fix concurrency bugs [44, 45], and for statistical debugging [18]. Those approaches often require failure points, error statements, or application instrumentation, which makes it more practical for detecting and diagnosing 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 and diagnosing 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 and diagnosis 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.

Our first study develops a hybrid approach to pinpoint the root cause functions of a performance anomaly in cloud systems. In the second study, we propose to conduct static analysis to detect data corruption related infinite loops in application functions which cause software hang in cloud systems. Finally, we plan to explore the auto-fix scheme to statically generate and dynamically evaluate patches for data corruption hang bugs.

1.3.1 Research Challenges

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

- **Online 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,

finding and fixing the root cause of the problem as quickly as possible is a top priority. As a result, any framework should be able to operate online, providing results in a matter of seconds or minutes as opposed to hours or days.

- **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).

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 describes the work related to our work.

Finally, Chapter 5 concludes our discussion of detecting and diagnosing performance bugs and describes future directions for our work.

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 [46] 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* [31, 33, 46, 64] 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* [8, 24]

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 [46, 64]. 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 [8, 24]. Unfortunately, without knowledge about program semantics, those dynamic techniques suffer from both false positives and false negatives too [8, 24].

2.1.1 A Performance Bug Example

To illustrate the challenge of performance bug diagnosis, we discuss Apache-37680¹ 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

¹We use “application name *dash* bug identifier” in the repository to denote each bug in this paper.

```

- if (ap_listeners && ap_listeners->next) {
+ use_nonblock = ap_listeners && ap_listeners->next;
  for (lr = ap_listeners; lr; lr = lr->next) {
    apr_status_t status;
    ...
    status = apr_socket_opt_set(lr->sd, APR_SO_NONBLOCK,
-                                     1);
+                                     use_nonblock);

    if (status != APR_SUCCESS) {
      ap_log_perror(APLOG_MARK, APLOG_STARTUP | APLOG_ERR, status, pool,
                    "ap_listen_open: unable to "
-                    + "make socket non-blocking");
+                    + "control socket non-blocking status");
    }
    return -1;
  }
- }

```

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, and “-” means the deleted lines.

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 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 [23, 24], performance counters [8], or other system metrics [58, 72]. 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 paper 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 [14, 55, 60, 78], 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 paper 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 [24] 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 [24]). None of these bugs can be covered by traditional pure static checkers that target on general software bugs (Infer [31], Findbugs [33]) or specific types of loop inefficiency bugs (Caramel [64]). 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 stand-alone 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.

```

if (fill_record_n_invoke_before_triggers (    thd,
      *info->update_fields, *info->update_values,
-      0,
+      info->ignore,
      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.

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 [42]). 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 `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


```

apr_bucket *e = APR_BRIGADE_FIRST(bb);
while (1) {
    ...
    if (APR_BUCKET_IS_EOS(e)) {
        ap_remove_output_filter(f);
        return ap_pass_brigade(f->next, bb);
    }
    if (APR_BUCKET_IS_METADATA(e)) {
+       e = APR_BUCKET_NEXT(e);
        continue;
    }
    ...
}

```

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.

```

if (len < 0) { ... return -1; }
else if (len == 0) { ... return -2; }
+ else { joblist_append(srv, con); }
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.

may return unexpected output. When those unexpected return values are not properly handled, the 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 `atoi` is called by `stream_reqbody_c1` to convert the `old_c1_val` string into the long integer, `c1_val`. This string happened to be larger than 2GB on the user's 32-bit machine. The integer overflow caused `atoi` 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 `atoi` with its large-file alternate: `apr_strtoff`.

R4: Unchanged loop exit condition variables. This rule looks for the loops whose exit condition 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 [57] for C/C++ applications and Findbugs [33] 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 [24] 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 [8, 9, 38, 52, 81], 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 [23, 78], 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 [51] to group those execution units that perform similar operations together based on the appearance vector feature. Next, we use the nearest neighbor algorithm [77] to perform outlier detection within each cluster to identify abnormal execution units. Frequent episode mining [1, 67] 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., [61]). 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 [60]. 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 [57]. 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 [19].

For Java applications, we implemented our transformations using Findbugs analysis infrastructure [33]. 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

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 <code>fopen</code> on file > 2 GB on 32-bit systems. | hang |
| Lighttpd-1212 | Keep processing same event when the return value <code>errno</code> 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 <code>INSERT DELAYED</code> statement but one of them has a locked table. | hang |
| MySQL-65615 | $5 \times$ slowdown in the table insertions after truncating a large table. | slowdown |
| Cassandra-5064 | <code>ALTER TABLE</code> command keeps flushing empty <code>Memtable</code> . | hang |
| HDFS-3318 | HDFS client keeps reading a > 2 GB file when the file length is represented by an <code>int</code> . | 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 |

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 |

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 [6] and Bugzilla [3].

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 paper, 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 [70], and tricky due to limited and often ambiguous information [47] 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 [24]. 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) [59], 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 [27] running 32-bit Ubuntu 12.04 kernel v3.2.0.

Our experiments use the same workloads as PerfScope [24] 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 [64], Findbugs [33], Infer [31] and PerfScope [24].

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.

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 | Hytrace-static | Coverage | | | | | # of bugs matched by each rule | | | | |
|-------------|--------------|----------------|-------------|---------------|----------------|------------------|---------|--------------------------------|----|----|----|----|
| | | | 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 |

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 2.2 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. 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 3.4.1).

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 |

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 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

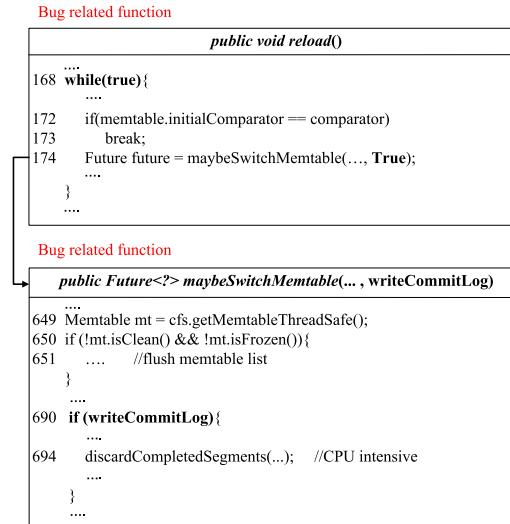


Figure 2.7 Partial call graph for Cassandra-5064 bug.

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.

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

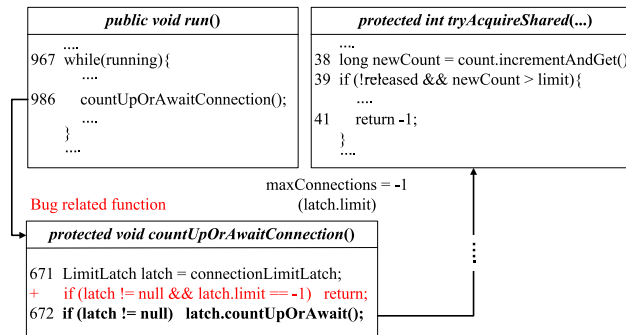


Figure 2.8 Partial call graph for Tomcat-53173 bug.

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 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 41 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

Bug related function

```
public void run()
+   try {
+       .... //runtime exception
+   finally {
193   this.appAggregationFinished.set(true);
+   }
```

Bug related function

```
public void join()
+   ....
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 }
```

Figure 2.9 Partial call graph for Mapreduce-3738 bug.

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 [20], 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.

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,`

Table 2.6 Performance of Hytrace-static program analysis and Hytrace-dynamic trace analysis (the run-time workload is described in Section 3.4.1).

| 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 |

`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 3.4.1. 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 [60], PCatch [56]) 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.

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

3.1 Introduction

Cloud server systems such as Hadoop and Cassandra [4, 5] 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 [35, 37] 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 [82].

Performance bugs¹ are notoriously difficult to debug because they typically produce little useful debugging information. The problem exacerbates in cloud server systems since the developer

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

```

//LeaseManager.java                                     #HDFS-4882(v0.23.0)
393 private synchronized void checkLeases() {
    ...
395 for (; sortedLeases.size() > 0; ) {
    ...
411 try { //p is a file's lease path
412     if (fsnamesystem.internalReleaseLease(
413         oldest, p, ...)) {
        ...//remove p from sortedLeases
416     }
        ...
420 } catch (IOException e) {
        ...//remove p from sortedLeases
423     }
        ...
429 }
430 }

```

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.

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., [11, 16, 35, 43, 80]) and performance bugs (e.g., [22, 40, 49, 74]), little research has been done to study the intersection 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 [22, 24, 25, 46].

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 3.1. 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 [39]. 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 paper 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 paper 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 [33] and Infer [31], 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 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 [25, 46], 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 [75] 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 dependent 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 [11, 12, 41, 48, 65, 71], software fault [80]) 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

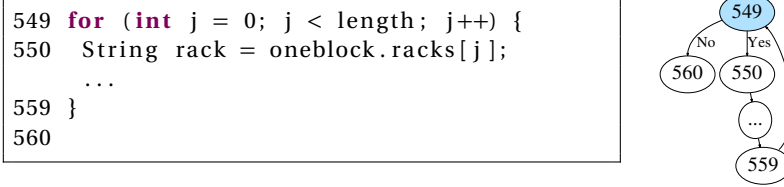


Figure 3.2 The example of a simple loop with the source code block and the corresponding CFG in the CombineFileInputFormat class in Hadoop v0.23.0.

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 the source code and its CFG of a simple loop². 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

²DScope 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 paper.

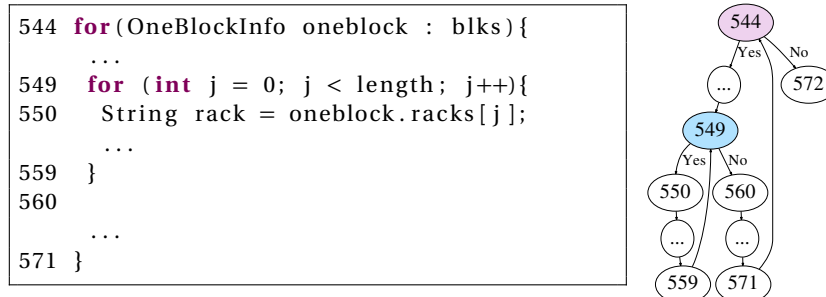


Figure 3.3 The example of nested loops with the source code block and the corresponding CFG in the CombineFileInputFormat class in Hadoop v0.23.0.

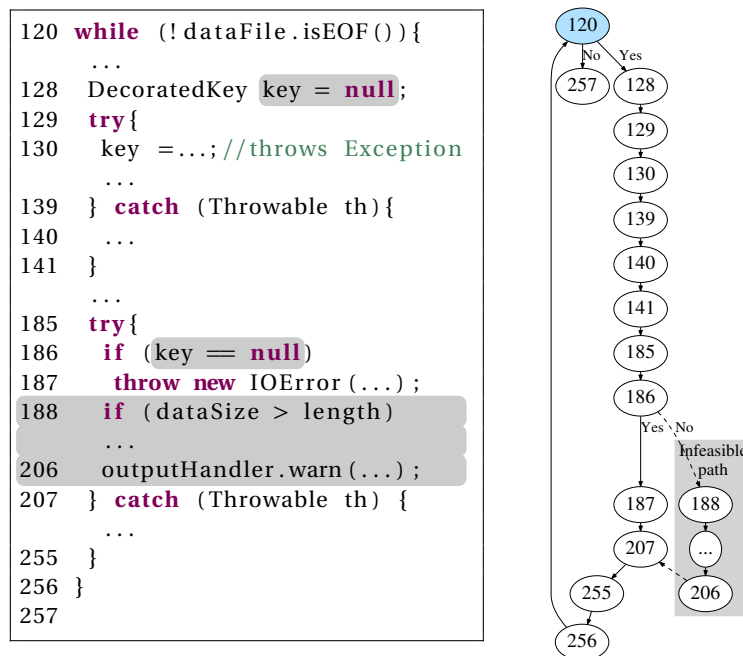


Figure 3.4 The example of a loop containing exception handling constructs with the source code block and the corresponding CFG in the Scrubber class in Cassandra v2.0.8.

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

```

//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`.

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* 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 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`


```

//Soot IR
3  if l8 >= 10 goto line #12 //l8 >= 10 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.

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 bytecode³. 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.

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

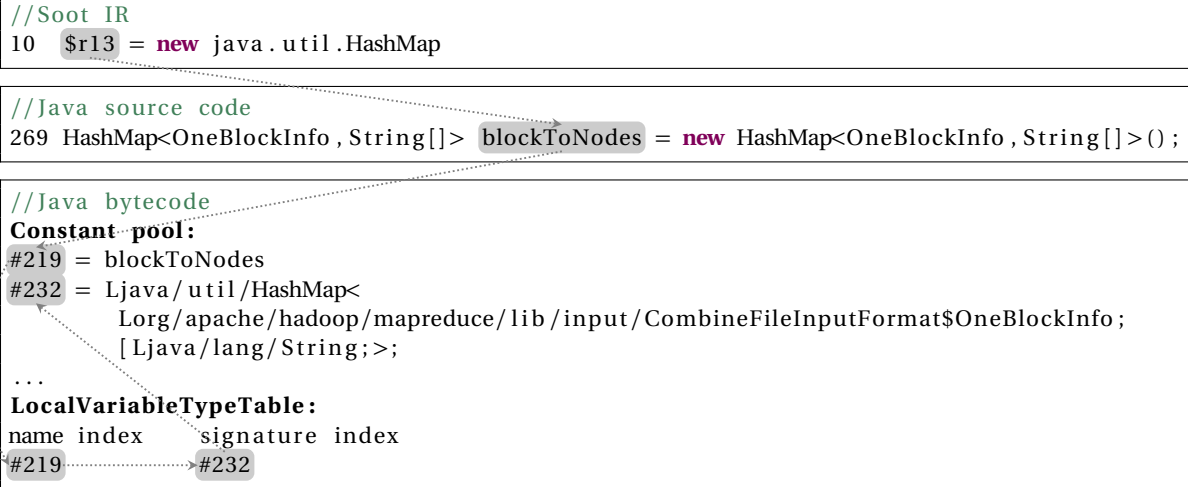


Figure 3.7 The `java.util.HashMap<K,V>` example in the `CombineFileInputFormat` class in Hadoop v0.23.0.

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 `$r13` is defined at line #269 in the source code with name `blockToNodes`. Next, DScope leverages Coffi [79], a Java bytecode parser, to extract the full type information for the target variable val_{src} . Specifically, the constant pool provides index lookup for each variable and `LocalVariableTypeTable` provides the mapping from the variable's index to the index of its signature which contains the full type information. In Figure 3.7, the constant pool indicates that the index of the variable `blockToNodes` is #219 and the `LocalVariableTypeTable` lookup tells its corresponding signature index is #232. DScope checks the constant pool using the index number #232 to derive the full type information of `blockToNodes`. Since `blockToNodes` consists of `Combine- FileInputFormat$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

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 |

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 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

```

//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.

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 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

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 |

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`.

```

//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.

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 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 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 §3.4). 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 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

```
//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;
190     }
191 }
```

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 /
191     ...
194     BUFFER_SIZE = conf.getInt(...); / BUFFER_SIZE = 0 /
195     ...
229 }
```

```
78 private void readLocalFile(Path path, ...) throws IOException {
79     ...
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);
88     }
89 }
```

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.

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, 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


```

//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(); iter.hasNext(); ) {
360         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);
                ...
            }
        }
    }
}

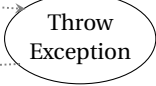
```

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.

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

```
//TestProcfsBasedProcessTree.java #Yarn-6991(v0.23.0)
//Thread #1
62 private class RogueTaskThread extends Thread {
63     public void run() {
64         try {
65             ...
72         args.add(" echo $$ > " + pidFile + "");
73         shexec = new ShellCommandExecutor(args ...);
74         shexec.execute();
65         ...
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()) {
90         ...
91         Thread.sleep(500);
92         ...
93     }
}
```

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.

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 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 loop 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 3.15. 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 `IOException` (line #187),

```

//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
...
139     } catch (Throwable th){
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.

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.

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.

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 |

3.4.1 Evaluation Methodology

DScope is implemented on top of Soot v2.5.0 [75], a Java bytecode analysis infrastructure, with the latest Coffi library [79], 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) [33] and Infer(v0.9.2) [31].

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 then search the bug repository (i.e., JIRA [6]) to see whether they are already reported. If they are,

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 | v0.23.0 | 2011/11/11 | 4 | 6 | 0 | 0 |
| Common | 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 |

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 schemes. 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 scope of this paper, 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 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

```

//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.

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).

Table 3.5 The detection comparison 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 |

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 |

CHAPTER

4

RELATED WORK

4.1 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. [46] employ rule-based methods to detect performance bugs that violate efficiency rules that have been violated before. Chen et al. [17] 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 [73], inefficient nested loops [63] and workload-dependent loops [84]. 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 [48] 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.

4.2 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 [9] uses symbolic execution to automatically identify and suggest fixes to performance bugs caused by configuration or input-based problems. Strider [81] 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 [26] focuses on differentiating external faults (e.g., interference from co-located applications) from internal faults (e.g., software bugs) for system performance anomalies. IntroPerf [52] automatically infers the latency of user-level and kernel-level function calls based on OS tracers. StackMine [38] 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. [2] identify the performance bottleneck nodes by conducting causal path analysis on the message-level traces (e.g., RPC message). Kasick et al. [50] 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. [85] and CloudSeer [86] 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).

4.3 Hybrid bug diagnosis

Hybrid techniques have been used to fix concurrency bugs. For example, AFix [44] and CFix [45] 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 [18] 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 [87] 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.

4.4 Data corruption study and detection

Previous work has been extensively studied the data corruption problems in storage systems. Hwang et al. [41] and Schroeder et al. [71] studied the data corruptions in memory devices. They found that DRAM failures occur more frequently than expected. Bairavasundaram et al. [11, 12] and Oleksenko et al. [65] 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 [15] detected file system corruption caused by storage hardware, e.g., latent sector errors. Fryer et al. [34, 76] 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).

4.5 Fault injection

Previous work [13, 36, 54] used fault injection techniques to analyze the failure behaviors (e.g., hang, crash) of both software and hardware systems. For example, HSFI [54] injected faults in the source code. Fault injection is also widely used to check whether file systems can handle certain type of data corruptions [32, 35, 69, 88]. For instance, Bairavasundaram et al. [10] used context aware fault

injections to find disk errors in virtual memory systems. Zhang et al. [88] 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 [35] 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.

4.6 Functional bug detection

Apart from performance bugs, recent works have also been done to detect functional bugs. pbSE [83] conducted concolic execution to detect functional bugs and generate test cases for those bugs. Kollenda et al. [53] 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.

4.7 Software testing

DeepXplore [68] 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 [66] 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. [29] 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

5

CONCLUSIONS AND FUTURE WORK

This report focuses on developing two key techniques for detecting and diagnosing performance bugs: diagnosing real-world performance bugs using a hybrid approach, and detecting data corruption hang bugs using static analysis. In future, we also plan to fix performance bugs especially data corruption hang bugs using both static patch generation and dynamic patch testing. These three motivate and complement each other. They first two have been proven to be efficient and effective for expediting the detection and diagnosis 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.

5.1 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).

5.2 Future Work

In this report, we have shown that our performance bug detection and diagnosis framework can effectively identify the root-cause functions and application infinite loops which cause performance problems in cloud systems.

We plan to explore auto-fix schemes to correct data corruption hang bugs based on their types, as described in §3.3.

- When the error code returned by the I/O operations directly affects the loop strides, one possible fix is to add extra error code checking statements in the loop exit conditions to avoid the software hang.
- If the corrupted data content indirectly affects loop strides, one possible fix could be adding additional check over the data content before it is used in the loop body.
- When the improper exception handling causes the loop stride update to be skipped, one possible fix is to add additional exception handling to properly update the stride when data corruption occurs.

Our auto-fix scheme can have one or more candidate patches for each bug. We tests these patches as follows.

- Control flow checking. We need to make sure the movement does not break control dependencies, causing a statement to execute for more or fewer times than it should be.

- Data flow checking. We need to make sure the movement does not break existing define-use data dependency within one thread, causing the patched software to deviate from the expected program semantic.
- Performance checking. We need to check whether the movement could bring risks of severe performance slowdowns and even infinite loops.

We finally report all the patches which pass above checking.

BIBLIOGRAPHY

- [1] Agrawal, R. & Srikant, R. “Fast algorithms for mining association rules”. *VLDB*. 1994.
- [2] Aguilera, M. K. et al. “Performance Debugging for Distributed Systems of Black Boxes”. *SOSP*. 2003.
- [3] *Apache Bugzilla*. <https://bz.apache.org/bugzilla/>.
- [4] *Apache Cassandra*. <http://cassandra.apache.org/>. 2018.
- [5] *Apache Hadoop*. <http://hadoop.apache.org/>. 2018.
- [6] *Apache JIRA*. <https://issues.apache.org/jira>.
- [7] *App Engine*. <https://cloud.google.com/appengine/>.
- [8] Arulraj, J. et al. “Production-run software failure diagnosis via hardware performance counters”. *ASPLOS*. 2013.
- [9] Attariyan, M., Chow, M. & Flinn, J. “X-ray: Automating root-cause diagnosis of performance anomalies in production software”. *OSDI*. 2012.
- [10] Bairavasundaram, L. N., Arpaci-Dusseau, A. C. & Arpaci-Dusseau, R. H. “Dependability Analysis of Virtual Memory Systems”. *DSN*. 2006.
- [11] Bairavasundaram, L. N. et al. “An Analysis of Data Corruption in the Storage Stack”. *TOS* **4.3** (2008), 8:1–8:28.
- [12] Bairavasundaram, L. N. et al. “Analyzing the Effects of Disk-Pointer Corruption”. *DSN*. 2008.
- [13] Barton, J. H. et al. “Fault Injection Experiments Using FIAT”. *TC* **39.4** (1990).
- [14] Bhatia, S. et al. “Lightweight, High-resolution Monitoring for Troubleshooting Production Systems”. *OSDI*. 2008.
- [15] Bonwick, J. & Moore, B. *ZFS—The Last Word In File Systems*. https://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf. 2007.
- [16] Borisov, N. et al. “Dealing Proactively with Data Corruption: Challenges and Opportunities”. *SMDB*. 2011.
- [17] Chen, T.-H. et al. “Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping”. *ICSE*. 2014.
- [18] Chilimbi, T. M. et al. “HOLMES: Effective Statistical Debugging via Efficient Path Profiling”. *ICSE*. 2009.
- [19] *Clang*. <https://clang.llvm.org/>. 2018.

- [20] Dai, T. et al. "Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures". *SoCC*. 2017.
- [21] Dai, T. et al. "DScope: Detecting Real-World Data Corruption Hang Bugs in Cloud Server Systems". *SoCC*. 2018.
- [22] Dai, T. et al. "Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures". *IEEE Transactions on Parallel and Distributed Systems* (2018).
- [23] Dean, D. J., Nguyen, H. & Gu, X. "UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems". *ICAC*. 2012.
- [24] Dean, D. J. et al. "PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures". *SOCC*. 2014.
- [25] Dean, D. J. et al. "Automatic Server Hang Bug Diagnosis: Feasible Reality or Pipe Dream?" *ICAC*. 2015.
- [26] Dean, D. J. et al. "PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds". *TPDS*. 2015.
- [27] Desnoyers, M. & Dagenais, M. R. "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux". *Linux Symposium*. 2006.
- [28] *EC2*. <https://aws.amazon.com/ec2/>.
- [29] Elia, I. A., Laranjeiro, N. & Vieira, M. "Test-Based Interoperability Certification for Web Services". *DSN*. 2015.
- [30] *Embarrassment*. <https://dom.as/2009/06/26/embarrassment/>.
- [31] *Facebook Infer*. <http://fbinfer.com/>.
- [32] Fiala, D. et al. "Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing". *SC*. 2012.
- [33] *Findbugs*. <http://findbugs.sourceforge.net/>. 2018.
- [34] Fryer, D. et al. "Checking the Integrity of Transactional Mechanisms". *TOS* **10.4** (2014), 17:1–17:23.
- [35] Ganesan, A. et al. "Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions". *FAST*. 2017.
- [36] Gu, W. et al. "Characterization of Linux Kernel Behavior Under Errors". *DSN*. 2003.
- [37] Gunawi, H. S. et al. "What Bugs Live in the Cloud?: A Study of 3000+ Issues in Cloud Systems". *SOCC*. 2014.
- [38] Han, S. et al. "Performance Debugging in the Large via Mining Millions of Stack Traces". *ICSE*. 2012.

- [39] *HDFS-4882*. <https://issues.apache.org/jira/browse/HDFS-4882>. 2013.
- [40] Huang, J., Zhang, X. & Schwan, K. “Understanding Issue Correlations: A Case Study of the Hadoop System”. *SOCC*. 2015.
- [41] Hwang, A. A., Stefanovici, I. A. & Schroeder, B. “Cosmic Rays Don’t Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design”. *ASPLOS*. 2012.
- [42] *Hytrace*. <http://dance.csc.ncsu.edu/projects/sysMD/Hytrace.html>.
- [43] Jiang, W. et al. “Is Disk the Dominant Contributor for Storage Subsystem Failures? A Comprehensive Study of Failure Characteristics”. *FAST*. 2008.
- [44] Jin, G. et al. “Automated atomicity-violation fixing”. *PLDI*. 2011.
- [45] Jin, G. et al. “Automated concurrency-bug fixing”. *OSDI*. 2012.
- [46] Jin, G. et al. “Understanding and Detecting Real-World Performance Bugs”. *PLDI*. 2012.
- [47] Jin, W. & Orso, A. “BugRedux: Reproducing Field Failures for In-house Debugging”. *ICSE*. 2012.
- [48] Kadav, A., Renzelmann, M. J. & Swift, M. M. “Tolerating Hardware Device Failures in Software”. *SOSP*. 2009.
- [49] Kaldor, J. et al. “Canopy: An End-to-End Performance Tracing And Analysis System”. *SOSP*. 2017.
- [50] Kasick, M. P. et al. “Black-box Problem Diagnosis in Parallel File Systems”. *FAST*. 2010.
- [51] Kaufman, L. & Rousseeuw, P. J. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009.
- [52] Kim, C. H. et al. “IntroPerf: Transparent Context-sensitive Multi-layer Performance Inference Using System Stack Traces”. *SIGMETRICS*. 2014.
- [53] Kollenda, B. et al. “Towards Automated Discovery of Crash-Resistant Primitives in Binary Executables”. *DSN*. 2017.
- [54] Kouwe, E. van der & Tanenbaum, A. S. “HSFI: Accurate Fault Injection Scalable to Large Code Bases”. *DSN*. 2016.
- [55] Kutare, M. et al. “Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers”. *ICAC*. 2010.
- [56] Li, J. et al. “PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems”. *EuroSys*. 2018.
- [57] *LLVM*. <http://llvm.org/>.
- [58] Mysore, R. N. et al. “Gestalt: Fast, Unified Fault Localization for Networked Systems”. *USENIX ATC*. 2014.

- [59] *NCSU Virtual Computing Lab*. <http://vcl.ncsu.edu/>.
- [60] Nguyen, H. et al. "FChain: Toward Black-box Online Fault Localization for Cloud Systems". *ICDCS*. 2013.
- [61] Nguyen, H. et al. "Insight: In-situ Online Service Failure Path Inference in Production Computing Infrastructures". *USENIX ATC*. 2014.
- [62] Nistor, A., Jiang, T. & Tan, L. "Discovering, Reporting, and Fixing Performance Bugs". *MSR*. 2013.
- [63] Nistor, A. et al. "Toddler: Detecting Performance Problems via Similar Memory-access Patterns". *ICSE*. 2013.
- [64] Nistor, A. et al. "Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes". *ICSE*. 2015.
- [65] Oleksenko, O. et al. "Efficient Fault Tolerance using Intel MPX and TSX". *DSN*. 2016.
- [66] Oleksenko, O. et al. "Fex: A Software Systems Evaluator". *DSN*. 2017.
- [67] Patnaik, D. et al. "Efficient Episode Mining of Dynamic Event Streams." *ICDM*. 2012.
- [68] Pei, K. et al. "DeepXplore: Automated Whitebox Testing of Deep Learning Systems". *SOSP*. 2017.
- [69] Prabhakaran, V. et al. "IRON File Systems". *SOSP*. 2005.
- [70] Saha, R. K., Khurshid, S. & Perry, D. E. "An Empirical Study of Long Lived Bugs". *CSMR-WCRE*. 2014.
- [71] Schroeder, B., Pinheiro, E. & Weber, W.-D. "DRAM Errors in the Wild: A Large-Scale Field Study". *SIGMETRICS*. 2009.
- [72] Shen, K. et al. "Reference-driven Performance Anomaly Identification". *SIGMETRICS*. 2009.
- [73] Song, L. & Lu, S. "Statistical Debugging for Real-World Performance Problems". *OOPSLA*. 2014.
- [74] Song, L. & Lu, S. "Performance Diagnosis for Inefficient Loops". *ICSE*. 2017.
- [75] *Soot: A Framework for Analyzing and Transforming Java and Android Applications*. <https://sable.github.io/soot/>. 2018.
- [76] Sun, K. et al. "Robust Consistency Checking for Modern Filesystems". *RV*. 2014.
- [77] Tan, P.-N., Steinbach, M. & Kumar, V. *Introduction to Data Mining*. Addison Wesley, 2005.
- [78] Tan, Y. et al. "PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems". *ICDCS*. 2012.

- [79] Verbrugge, C. *Using Coffi*. <http://www.sable.mcgill.ca/~clump/Coffi/Coffi.ps>. 1996.
- [80] Wang, P., Dean, D. J. & Gu, X. "Understanding Real World Data Corruptions in Cloud Systems". *IC2E*. 2015.
- [81] Wang, Y.-M. et al. "Strider: A black-box, state-based approach to change and configuration management and support". *LISA*. 2003.
- [82] *What Lessons can be Learned from BA's Systems Outage?* <http://www.extraordinarymanagementservices.com/news/what-lessons-can-be-learned-from-bas-systems-outage/>. 2017.
- [83] Xiao, Q. et al. "pbSE: Phase-Based Symbolic Execution". *DSN*. 2017.
- [84] Xiao, X. et al. "Context-sensitive Delta Inference for Identifying Workload-dependent Performance Bottlenecks". *ISSTA*. 2013.
- [85] Xu, W. et al. "Detecting Large-scale System Problems by Mining Console Logs". *SOSP*. 2009.
- [86] Yu, X. et al. "CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs". *ASPLOS*. 2016.
- [87] Zamfir, C. & Candea, G. "Execution Synthesis: A Technique for Automated Software Debugging". *EuroSys*. 2010.
- [88] Zhang, Y. et al. "End-to-End Data Integrity for File Systems: A ZFS Case Study". *FAST*. 2010.