Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures

Ting Dai, *Member, IEEE,* Daniel Dean, *Member, IEEE,* Peipei Wang, *Member, IEEE,* Xiaohui Gu, *Senior Member, IEEE,* and Shan Lu, *Senior Member, IEEE*

Abstract—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. In this paper, 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.

Index Terms—Static analysis; Dynamic analysis; Reliability, availability, and serviceability; Debugging aids; Performance

1 Introduction

Cloud computing infrastructures [3], [5] have become increasingly popular by allowing users to access computing resources in a cost-effective way. However, 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 [23] 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* [6], [7], [23], [35] 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* [13], [19] 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 [23], [35]. 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 [13], [19]. Unfortunately, without knowledge about program semantics, those dynamic techniques suffer from both false positives and false negatives too [13], [19].

1.1 A Performance Bug Example

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

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

- *Ting Dai, Peipei Wang, and Xiaohui Gu are with the North Carolina State University.*
  E-mail: {tdai,pwang7}@ncsu.edu, gu@csc.ncsu.edu
- *Daniel Dean is with the InsightFinder Inc.*
  E-mail: daniel@insightfinder.com
- *Shan Lu is with the University of Chicago.*
  E-mail: shanlu@cs.uchicago.edu

1. We 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;
      }
    }
- }
```

Fig. 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.

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

It is challenging to precisely detect the above problem using pure static checking. A rule that precisely captures the root cause of this bug is that blocking calls should not be made on a socket configured to not allow blocking calls. This rule is almost infeasible to check statically — the socket configuration can happen long before the blocking call, and inter-procedural path-sensitive static analysis cannot scale to complicated production server software. Furthermore, even if this rule is checkable, it is too specific. Providing good diagnosis coverage using such specific rule checking is difficult if not totally impossible. Note that, a traditional generic infinite loop detector would not 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 [18], [19], performance counters [13], or other system metrics [32], [39]. Unfortunately, for bugs like the one in Figure 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.

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 [15], [30], [34], [42], 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 [19] 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 [19]). None of these bugs can be covered by traditional pure static checkers that target on general software bugs (Infer [6], Findbugs [7]) or specific types of loop inefficiency bugs (Caramel [35]). 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.

The rest of the paper is organized as follows. Section 2 describes the design of the Hytrace system. Section 3 presents our implementation of Hytrace. Section 4 talks about the methodology. Section 5 shows the experimental evaluation. Section 6 discusses the limitation and future work of Hytrace. Section 7 compares our work with related work. Finally, the paper concludes in Section 8.

2 DESIGN

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

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.

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.

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

Fig. 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.

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 [8]). 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 uses a hard-coded constant value of 0, which causes MySQL to never ignore errors when executing the `fill_record_n_invoke_before_triggers` function. In most cases, this is not a problem as errors would be handled appropriately (e.g., logged). However, in certain circumstances, such as, when executing the `INSERT IGNORE` command, errors should be ignored but are not, which causes the system to hang.

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

R2: Null parameter function calls. A function call that uses `null` as a pointer/object parameter matches this rule; the function that issues such a null-parameter function call will be considered as a candidate faulty function. This type of function calls can be related to performance problems in several ways. The `null` parameter is sometimes

```
  AllocateRequest allocateRequest =
    AllocateRequest.newInstance(   lastResponseID,
                    super.getApplicationProgress(),
                new ArrayList<ResourceRequest>(ask),
                new ArrayList<ContainerId>(release),
-                                          null);
+                                      blacklistReq);
```

Fig. 3. Example for R2: null parameter (Mapreduce-5489 bug). The bug occurs as a result of not using node blacklisting feature in Resource-Manager requests, hanging MapReduce jobs.

```
-   cl_val = atol(old_cl_val);
+   if (APR_SUCCESS != (status = apr_strtoff(
+                   &cl_val, old_cl_val, NULL, 0))) {
+     return status;
+   }
    ...
    while (!APR_BUCKET_IS_EOS(...input_brigade)){
      ...
      bytes_streamed += bytes;
      ...
      if (bytes_streamed > cl_val)
        continue;
      ...
      //input_brigade is changed after
    }
```

Fig. 4. Example for R3: unsafe function (Apache-40883 bug). The bug occurs as a result of calling `atol` to convert a string, whose value is greater than 2 GB, into a long integer in `stream_reqbody_cl` function, hanging Apache system.

unexpected and hence not properly handled, leading to unexpected execution behavior. Sometimes, the `null` is the default parameter. When developers "lazily" use a default parameter, inefficiency may follow. Figure 3 shows an example for this rule. In this bug, `null` is the default value for the last parameter of `AllocateRequest::newInstance` method. Using the default value causes the Resource-Manager to not blacklist any bad Node-Managers during job allocation, even when a bad Node-Manager is already blacklisted by the Application-Master. As a result, the Resource-Manager could keep allocating the same blacklisted Node-Manager to the Application-Master, leading to a hang problem.

R3: Unsafe function calls. Some widely used I/O library functions, such as `atol` and `fopen`, may return unexpected output. When those unexpected return values are not properly handled, the 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 4 shows the patch for Apache-40883 bug. In this bug, an unsafe function `atol` is called by `stream_reqbody_cl` to convert the `old_cl_val` string into the long integer, `cl_val`. This string happened to be larger than 2GB on the user's 32-bit machine. The integer overflow caused `atol` to return 0, which in turn caused the if branch be taken in every iteration of the while loop. Once that happens, a `continue` statement is executed without updating `input_brigade` and then goes to the loop header, the same `input_brigade` value makes the condition of while loop always be true, causing whole Apache to hang. The patch simply replaced `atol` with its large-file alternate: `apr_strtoff`.

```
  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;
      }
      ...
  }
```

Fig. 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;
```

Fig. 6. Example for Rule 5: uncovered branch (Lighttpd-2197 bug). The bug occurs as a result of unhandling fragmented ssl request case, stalling Lighttpd.

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 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 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 [9] for C/C++ applications and Findbugs [7] 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 3.

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 [19] 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 [13], [14], [22], [29], [43], 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 [18], [42], 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 [28] to group those execution units that perform similar operations together based on the appearance vector feature. Next, we use the nearest neighbor algorithm [41] to perform outlier detection within each cluster to identify abnormal execution units. Frequent episode mining [11], [37] 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., [33]). If a function has multiple appearances in the final buggy function list, we only keep its highest rank.

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 moni-

toring 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 [34]. 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.

3 Implementation

To perform static analysis for C/C++ applications, we have developed code analysis passes using LLVM [9]. 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 [4].

For Java applications, we implemented our transformations using Findbugs analysis infrastructure [7]. 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.

TABLE 1
Descriptions of the 14 real-world bugs we reproduced.

Bug name	Root-cause description	Symptom
Apache-37680	Make a blocking "accept" call with non-blocking configuration.	hang
Apache-43238	Set up new connections with non-`keep-alive` configuration.	slowdown
Apache-45856	Call `fopen` on file $>$ 2 GB on 32-bit systems.	hang
Lighttpd-1212	Keep processing same event when the return value `errno` is mishandled.	hang
Lighttpd-1999	Keep reading and discarding response data while processing header information.	slowdown
Memcached-106	Keep reading a non-existent package when the previous packages overwrite the read buffer.	hang
MySQL-54332	Two threads execute the `INSERT DELAYED` statement but one of them has a locked table.	hang
MySQL-65615	5 × slowdown in the table insertions after truncating a large table.	slowdown
Cassandra-5064	`ALTER TABLE` command keeps flushing empty `Memtable`.	hang
HDFS-3318	HDFS client keeps reading a $>$ 2 GB file when the file length is represented by an `int`.	hang
Mapreduce-3738	Endless wait for an atomic variable to be set.	hang
Tomcat-53450	Tomcat tries to upgrade a read lock to a write lock.	hang
Tomcat-53173	Keep dropping incomming 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

4 Evaluation Methodology

Our experimental evaluation uses 133 real-world performance bugs: 53 C/C++ performance bugs from 5 server applications (Apache http web server, Lighttpd web server, Memcached distributed memory caching system, MySQL database engine, and Squid web proxy) and 80 Java performance bugs from 4 server applications (Cassandra distributed key-value store, Hadoop MapReduce distributed computing infrastructure, HDFS distributed file system, and Tomcat application server). Those bugs are collected by searching for the terms hangs, *100% CPU*, *stuck*, *slowdown* and *performance* in JIRA [2] and Bugzilla [1].

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.

TABLE 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

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 [38], and tricky due to limited and often ambiguous information [26] 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 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 1 follow the similar statistics. In addition, these 14 bugs include all the benchmarks in the PerfScope paper [19]. 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) [10], 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 [21] running 32-bit Ubuntu 12.04 kernel v3.2.0.

Our experiments use the same workloads as PerfScope [19] 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 [35], Findbugs [7], Infer [6] and PerfScope [19].

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.

5.1 Coverage and Precision Results

Table 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 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. 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. In contrast, Hytrace's static patterns favor generality over specification,

TABLE 3
Coverage comparison for all performance bugs and the matching frequency of each Hytrace static rule. "perf.": using only performance-related patterns/rules in Infer and Findbugs; "all": using all patterns/rules in Infer and Findbugs. "-": not supporting applications in specific languages (Caramel and Findbugs), or not implemented by Hytrace static (R3: unsafe function only checks C library functions).

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

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 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 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, 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 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 3 is similar to the average number of false positive functions for the 14 bugs in Table 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 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 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-

TABLE 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

consuming (Section 4).

Table 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.

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 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.

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. There is one case in this section and four additional cases in the online supplementary material due to space limitation.

Apache-37680 (C/C++): The patch and the cause of this bug (Figure 1) is already discussed in Section 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 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 1.1, the patch exactly changes the "constant parameter", 1, passed to the invocation of `apr_socket_opt_set` in function `ap_setup_listeners`.

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 6, it can finish analyzing hundreds of mega-bytes of traces usually within a couple of minutes.

TABLE 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 w/							
					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

TABLE 6
Performance of Hytrace-static program analysis and Hytrace-dynamic trace analysis (the run-time workload is described in Section 4).

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

The core part of the Hytrace trace analysis, frequent episode mining, can be easily parallelized and achieve much better performance, if needed. Note that, our evaluation uses workloads described in Section 4. To trigger the performance problems under diagnosis, we could have used shorter-running workloads, which would take less analysis time for Hytrace-dynamic.

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 [34], PCatch [31]) 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 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.

7 Related Work

Static rule-based bug detection. Much work has been done to develop static bug detection tools. Each work uses predefined heuristics/rules to specifically target certain types of performance bugs. Jin et al. [23] employ rule-based methods to detect performance bugs that violate efficiency rules that have been violated before. Chen et al. [16] 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 [40], inefficient nested loops [36] and workload-dependent loops [44]. 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 Section 1.

On-site bug diagnosis. Dynamic analysis techniques have been used to identify and fix performance bugs that are triggered in production environments. X-ray [14] uses symbolic execution to automatically identify and suggest fixes to performance bugs caused by configuration or input-based problems. Strider [43] 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 [20] focuses on differentiating external faults (e.g., interference from co-located applications) from internal faults (e.g., software bugs) for system performance anomalies. IntroPerf [29] automatically infers the latency of user-level and kernel-level function calls based on OS tracers. StackMine [22] 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. [12] identify the performance bottleneck nodes by conducting causal path analysis on the message-level traces (e.g., RPC message). Kasick et al. [27] 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. [45] and CloudSeer [46] 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).

Hybrid bug diagnosis. Hybrid techniques have been used to fix concurrency bugs. For example, AFix [24] and CFix [25] 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 [17] 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 [47] 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.

8 Conclusion

In this paper, 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.

Acknowledgments

This work was sponsored in part by NSF CNS1513942, and NSF CNS1149445. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF or U.S. Government. The authors would like to thank the anonymous reviewers for their insightful feedback and comments during the writing of this paper.

References

[1] Apache Bugzilla. https://bz.apache.org/bugzilla/.
[2] Apache JIRA. https://issues.apache.org/jira.
[3] App Engine. https://cloud.google.com/appengine/.
[4] Clang. https://clang.llvm.org/.
[5] EC2. https://aws.amazon.com/ec2/.
[6] Facebook Infer. http://fbinfer.com/.
[7] Findbugs. http://findbugs.sourceforge.net/.
[8] Hytrace. http://dance.csc.ncsu.edu/projects/sysMD/Hytrace.html.
[9] LLVM. http://llvm.org/.
[10] NCSU Virtual Computing Lab. http://vcl.ncsu.edu/.
[11] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.
[12] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.
[13] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. Production-run software failure diagnosis via hardware performance counters. In *ASPLOS*, 2013.
[14] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.
[15] Sapan Bhatia, Abhishek Kumar, Marc E. Fiuczynski, and Larry Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *OSDI*, 2008.
[16] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE*, 2014.
[17] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, 2009.
[18] Daniel J. Dean, Hiep Nguyen, and Xiaohui Gu. UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *ICAC*, 2012.
[19] Daniel J. Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. PerfScope: Practical online server performance bug inference in production cloud computing infrastructures. In *SOCC*, 2014.

[20] Daniel J. Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu, Anca Sailer, and Andrzej Kochut. PerfCompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds. In *TPDS*, 2015.

[21] Mathieu. Desnoyers and Michel R. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Linux Symposium*, 2006.

[22] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, 2012.

[23] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, 2012.

[24] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[25] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *OSDI*, 2012.

[26] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *ICSE*, 2012.

[27] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-box problem diagnosis in parallel file systems. In *FAST*, 2010.

[28] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009.

[29] Chung Hwan Kim, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Xiangyu Zhang, and Dongyan Xu. Introperf: Transparent context-sensitive multi-layer performance inference using system stack traces. In *SIGMETRICS*, 2014.

[30] Mahendra Kutare, Greg Eisenhauer, Chengwei Wang, Karsten Schwan, Vanish Talwar, and Matthew Wolf. Monalytics: Online monitoring and analytics for managing large scale data centers. In *ICAC*, 2010.

[31] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi Gunawi, Xiaohui Gu, Dongsheng Li, and Xicheng Lu. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In *EuroSys*, 2018.

[32] Radhika Niranjan Mysore, Ratul Mahajan, Amin Vahdat, and George Varghese. Gestalt: Fast, unified fault localization for networked systems. In *USENIX ATC*, 2014.

[33] Hiep Nguyen, Daniel J. Dean, Kamal Kc, and Xiaohui Gu. Insight: In-situ online service failure path inference in production computing infrastructures. In *USENIX ATC*, 2014.

[34] Hiep Nguyen, Zhiming Shen, Yongmin Tan, and Xiaohui Gu. FChain: Toward black-box online fault localization for cloud systems. In *ICDCS*, 2013.

[35] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, 2015.

[36] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, 2013.

[37] Debprakash Patnaik, Srivatsan Laxman, Badrish Chandramouli, and Naren Ramakrishnan. Efficient episode mining of dynamic event streams. In *ICDM*, 2012.

[38] Ripon K. Saha, Sarfraz Khurshid, and Dewayne E. Perry. An empirical study of long lived bugs. In *CSMR-WCRE*, 2014.

[39] Kai Shen, Christopher Stewart, Chuanpeng Li, and Xin Li. Reference-driven performance anomaly identification. In *SIGMETRICS*, 2009.

[40] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, 2014.

[41] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

[42] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. PREPARE: Predictive performance anomaly prevention for virtualized cloud systems. In *ICDCS*, 2012.

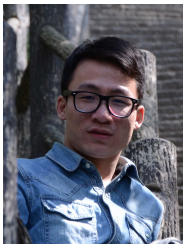
[43] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA*, 2003.

[44] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA*, 2013.

[45] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.

[46] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In *ASPLOS*, 2016.

[47] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys*, 2010.



Ting Dai is a Ph.D. student in the Department of Computer Science at North Carolina State University. He received a B.S. in information security and M.S. in computer software and theory from Nanjing University of Posts and Telecommunications, China in 2011 and 2014 respectively. He has interned with InsightFinder Inc. in the summer of 2016. Ting is a student member of the IEEE.

Daniel J. Dean is the V.P. of Analytics in InsightFinder Inc. He received his Ph.D. degress in 2015 from the Department of Computer Science at North Carolina State University. He received a B.S. and M.S. in computer science from Stony Brook University, New York in 2007 and 2009 respectively. He was a research staff member at IBM Research. Daniel is a member of the IEEE.

Peipei Wang is a Ph.D. student in the Department of Computer Science at North Carolina State University. Peipei received a B.S. in Software Engineering and M.S. in Computer Science from Xi'an Jiaotong University, China in 2010 and 2013 respectively. Peipei was a summer intern at Morgan Stanley IT Department in 2012. Peipei is a student member of the IEEE.

Xiaohui Gu is an Associate Professor in the Department of Computer Science at the North Carolina State University. She received her Ph.D. degree in 2004 and M.S. degree in 2001 from the Department of Computer Science, University of Illinois at Urbana-Champaign. She received her B.S. degree in computer science from Peking University, Beijing, China in 1999. She was a research staff member at IBM T. J. Watson Research Center, Hawthorne, New York, between 2004 and 2007. She received ILLIAC fellowship, David J. Kuck Best Master Thesis Award, and Saburo Muroga Fellowship from University of Illinois at Urbana-Champaign. She also received the IBM Invention Achievement Awards in 2004, 2006, and 2007. She has filed nine patents, and has published more than 60 research papers in international journals and major peer-reviewed conference proceedings. She is a recipient of NSF Career Award, four IBM Faculty Awards 2008, 2009, 2010, 2011, and two Google Research Awards 2009, 2011, best paper awards from ICDCS 2012 and CNSM 2010, and NCSU Faculty Research and Professional Development Award. She is a Senior Member of IEEE.

Shan Lu is an Associate Professor in the Department of Computer Science at the University of Chicago. She received her Ph.D. at University of Illinois, Urbana-Champaign, in 2008. Her research focuses on software reliability and software efficiency. She is a recipient of Alfred P. Sloan Research Fellow and NSF Career Award. Her co-authored papers won best paper and extinguished paper awards from OSDI 2016, ICSE 2015, FSE, 2014, FAST 2013, ACM-SIGPLAN 2011, and IEEE-Micro 2006. Shan currently serves as the Vice Chair of ACM-SIGOPS and the Associate Editor for IEEE Computer Architecture Letters.