# Automatic Server Hang Bug Diagnosis: Feasible Reality or Pipe Dream?

Daniel J. Dean, Peipei Wang, Xiaohui Gu, William Enck, and Guoliang Jin
North Carolina State University
Raleigh, North Carolina 27613
Email: {djdean2,pwang7}@ncsu.edu, {gu,enck}@csc.ncsu.edu, guoliang_jin@ncsu.edu

***Abstract*—It is notoriously difficult to diagnose server hang bugs as they often generate little diagnostic information and are difficult to reproduce offline. In this paper, we present a characteristic study of 177 real software hang bugs from 8 common open source server systems (i.e., Apache, Lighttpd, MySQL, Squid, HDFS, Hadoop Mapreduce, Tomcat, Cassandra). We identify three major root cause categories (i.e., programmer errors, mishandled values, concurrency issues). We then describe two major problems (i.e., false positives and false negatives) while applying existing rule-based bug detection techniques to those bugs.**

***Keywords*—hang bugs, characteristic study, performance**

## I. Introduction

When software hang bugs occur in production servers, millions of users can be affected. For example, in August of 2013, a 25-minute service outage of the Amazon Web Services [2] brought down many popular web sites such as Vine, Instagram, and Flipboard as well as Amazon itself [19]. It is notoriously difficult to diagnose those server hang bugs. On one hand, those hang problems often produce little diagnostic information (e.g., error messages) to help the developer to debug them. On the other hand, it is often difficult to reproduce those server hang problems for offline debugging.

### A. Summary of the State of the Art

A recent study [27] finds that poor error handling was the major cause for many failures in distributed data-intensive systems (e.g., Cassandra, Hadoop Mapreduce). HangWiz [23] automatically identifies hang bugs using static analysis to find the intersection between functions which block for some period of time (e.g., `connect`) and functions expected to complete in a timely fashion (e.g., UI update functions). However, our study shows that the root cause of server hang bugs are not limited to those identified problems. Previous work [6], [13], [25] have developed rule-based or heuristic-based static checkers to identify software bugs. However, these tools all rely on the premise that simple bug identification rules can be defined using a-priori information (e.g., existing patches) and then statically checked for bug detection. When such rules are difficult to define, it is challenging to adapt those rule-based techniques for automatic server hang bug diagnosis. A flurry of research work [26], [22], [21], [17], [24], [7] has been done to automatically identify and fix concurrency bugs using static analysis. Although these techniques are effective at identifying and fixing concurrency bugs, these types of techniques cannot be used to automatically identify or fix many of the hang bugs we studied.

### B. Our Contributions

In this paper, we present a characteristic study of 177 software hang bugs affecting 8 commonly used open source server systems (i.e., Apache, Lighttpd, MySQL, Squid, HDFS, Hadoop Mapreduce, Tomcat, Cassandra). We have identified three major categories which can effectively describe the root causes of those hang bugs. We describe in detail the types of bugs we assign to each category along with examples of the types of fixes that are typically applied to the bugs in each category. Specifically, this paper makes the following contributions:

- We identify that *programmer errors*, *mishandled values*, and *concurrency issues* are the three major reasons why software hangs occur in server systems.
- We find that static bug detection rules extracted from 81% of the studied hang bugs would generate *false positives* when used to find new bugs in either the same system or different systems. This makes it difficult to apply existing static analysis tools to diagnose server hang bugs.
- We discover that rules extracted from 64% of the studied hang bugs would be specific to a particular bug or software version. This causes *false negatives* in cases where bug variations or code changes are present.

Although there are issues (i.e., false positives, false negatives) with applying existing bug analysis schemes to the bugs we have studied, we believe automatic hang bug diagnosis is both necessary (as manual diagnosis is difficult and time consuming) and possible (after combining static rule checkers with runtime bug inference hints [9], [10], [8]. The aim of this study is to aid the development of automatic diagnosis tools by giving insight into different common causes of these bugs, helping to identify how to develop new techniques for mitigating these issues. For example, knowing incorrect timeout values are a common cause of hang bugs in many applications can enable the development of new tools designed only to target a subset of functions at runtime. In this case, runtime hints could indicate that a blocking `connect` function call without a timeout is taking longer than usual to complete, which would allow self-healing tools to automatically kill or close the connection. We also hope the observations of our study could help developers avoid hang bugs when developing new applications.

TABLE I: The applications we studied along with the number of bugs we examined.

| System Name | Type | Language (LOC) | # of bugs |
|---|---|---|---|
| Apache | web server | C/C++ (161K) | 13 |
| Lighttpd | web server | C (38K) | 6 |
| MySQL | database | C/C++ (1M) | 45 |
| Squid | proxy/web cache | C/C++ (180K) | 18 |
| HDFS | file system | Java (153K) | 21 |
| Hadoop MapReduce | data processing framework | Java (1.3M) | 35 |
| Tomcat | web server/servlet container | Java (285K) | 10 |
| Cassandra | database | Java (168K) | 29 |

The rest of this paper is organized as follows. We first present the methodology of our study in Section II. We then present our characteristic study in Section III. Next, we describe the issues we have identified with applying existing rule-based static analysis techniques to the bugs we studied in Section IV. Finally, we conclude the paper in Section V.

## II. Methodology

In this section we describe the methodology of our study. We begin by discussing how we collected the 177 hang bugs. Next, we briefly describe the root cause categories for classifying those hang bugs.

### A. Bug Collection

We chose eight commonly used open source systems to study: Apache [3], Lighttpd [16], MySQL [18], Squid [20], HDFS [12], Hadoop MapReduce [11], Tomcat [5], and Cassandra [4]. Each of these systems has well maintained bug reporting databases where users can report bugs they have encountered. As shown in Table I, these systems range from distributed databases to web-servers to web-proxies. All of those systems are and well maintained mature server systems, consisting of 38K to 1.3 million lines of code.

The bug reporting databases for each system we studied vary in implementation. However, each of the bug reporting databases allow users to search for bugs using search strings. To identify issues, we searched each of the databases for relevant hang bug terms. Specifically, we searched for the terms *hangs, 100% CPU, stuck, and performance*. We used the developer provided tags to search the MySQL database as well. We only considered bug reports which were closed with a resolution of being fixed. In addition, the majority of the bugs were marked as high priority bugs with tags such as "critical".

The search we performed resulted in several thousand matched bug reports. We manually examined the title and bug descriptions of those bugs to identify real hang bugs. We also removed any bug that did not have any root-cause details provided (e.g., patch). In particular, this caused us to remove many Lighttpd bugs as details are commonly omitted from those bug reports. After this initial filtering, we were left with 177 bugs, which we then studied in detail by reading the comments along with any provided patches.

Our bug collection was done without any bias across the eight systems we studied. In addition, we chose these systems to study as we feel they represent a good mix of the different types of server-side software. The results of our study are consistent across the eight systems. However, we do not claim by any means that our study is exhaustive and has studied all existing bugs for each system.

### B. Root Cause Categories

Based on our analysis of the 177 hang bugs, we have identified three major categories to classify the root causes of those hang bugs. We then divide each general category into different sub-categories to further classify each bug. The general categories indicate how the error is introduced to the code while the sub-categories describe common root causes.

- **Programmer error:** Each bug assigned to this category represents a situation involving programmer-related issues. These issues range from failing to set a timeout value for a function call to API misunderstanding. We further divide this category into three sub-categories: *1) Incorrect timeout* bugs involving either a timeout or wait value which is either missing or set incorrectly; *2) API misuse* bugs resulting from developers using a function or variable in a way which it was not intended; and *3) missing function call(s)* bugs resulting from one or more missing function calls.
- **Mishandled values:** The bugs assigned to this category represent situations in which a value is mishandled in some way. This can include situations in which a function returns an unexpected value or where an error handler is incomplete. We further divide this category into five sub-categories: *1) incorrect conditional check* bugs resulting from either a missing conditional check or a conditional check which does not perform as expected; *2) improper error handling* bugs involving cases in which an error condition is handled incorrectly; *3) unanticipated return value* bugs occurring when a function returns a value which the developers did not expect; *4) unanticipated input* bugs occurring when an unexpected input causes the system to hang; and *5) incorrect value definition* bugs resulting from a system using an undefined or incorrectly defined value.
- **Concurrency:** Each bug assigned to this category caused an application hang problem due to a concurrency issue. We divide the bugs in this category into two sub-categories: *1) deadlock* bugs involving situations where two operations are waiting on each other to complete; and *2) race condition* bugs involving a situation where a series of concurrent commands cause an application to hang.

Although most of the hang bugs are only assigned to a single category, those categories are not mutually exclusive.

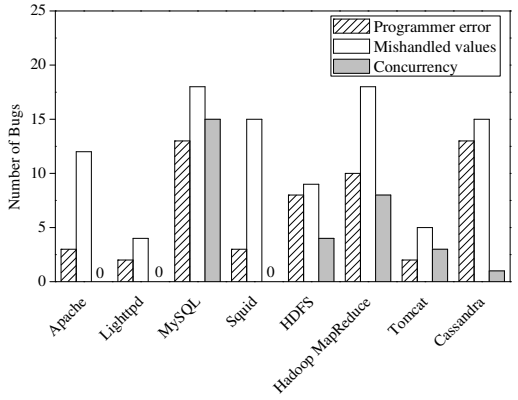


Fig. 1: Statistics on the 177 real hang bugs we examined.

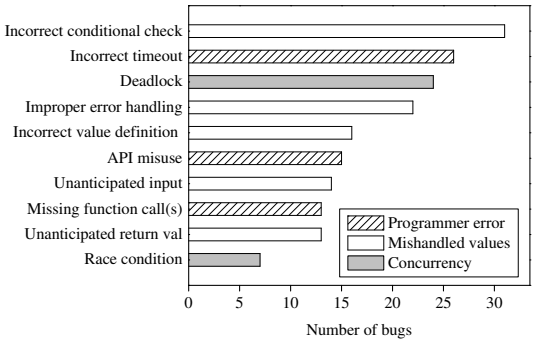


Fig. 2: The statistics of the bugs assigned to each sub-category.

For example, an improper error handling bug can occur as the result of an unanticipated return value.

## III. Root Cause Study

Our root cause study focuses on two issues: *why* a server hang problem occurred? and *how* are those hang problems fixed? In this section, we first present an overview of how the different server systems we examined were affected. We then discuss the three major categories of programmer error, mishandled values, and concurrency, in detail.

### A. Overview

Figure 1 shows the bug categories along with the number of bugs for each system in each category. We observe that not all the systems were affected equally by different types of bugs. We believe that this is due to the role of the different systems. For example, we found race condition bugs affected Hadoop MapReduce more than the other systems. This is likely due to the distributed nature of the system coupled with its complexity and typical deployment scale (i.e., 1000s of nodes).

Figure 2 shows the overall statistics of the bugs in each of the sub-categories ranked in descending order. As shown, the incorrect conditional check, timeout, and deadlock sub-categories have the most bugs assigned to them. Out of these, the incorrect timeout value finding was surprising, indicating that developers often do not expect certain calls to fail to complete.

### B. Programmer error

**Incorrect timeout:** We identified 26 bugs in this category. Each of these bugs is the result of a timeout value being either missing or incorrectly set on operation. The operations are typically blocking operations (e.g., `connect`) but can also be non-blocking commands (e.g., UI update functions).

For the bugs where a timeout value is missing, the reported issue is typically a hung server under certain conditions (e.g., specific request). This is usually because the buggy function typically expects the requested operation to always complete. For example, HDFS-1490[1] describes a case where a missing timeout value on a blocking socket call causes the system to hang indefinitely. The fix for this problem was simply to ensure a timeout value was set on the blocking call.

**API misuse:** We identified 15 bugs in this category. Each of these bugs occurs as the result of the developer's misunderstanding about which flags or functions to use.

An example of how these bugs can cause the system to hang is shown by Squid-3084. In this bug, there was a misunderstanding of which function call to use when handling an IPv6 address. Using the incorrect function call with an IPv6 address causes the connection to hang. As with the other examples in this category, the fix was to correct the misunderstanding (i.e., use the correct function call).

**Missing function call(s):** We identified 13 bugs in this category. These bugs are the result of a missing function call causing the system to hang for a period of time.

A simple example of this is Cassandra-5635. In this bug, the system hangs when a call setting the thread to run as a daemon was omitted. When a stop command is issued and the system is not running as a daemon, it will not shut down all threads because it expects more commands to be interactively issued. The fix was to add the missing function call.

**Observation 1** *Simple issues caused by programmers misunderstanding about how system components should be used were responsible for hangs in all the systems we tested.*

The most surprising thing we found in this category was the number of bugs that occurred as a result of a timeout value being either missing or incorrect. Applying existing techniques to these types of problems is difficult due to false positives. For example, setting a timeout value to all functions may not be correct for long running functions. Additionally, we found the other sub-categories are susceptible to false negatives because they require developer knowledge to find and fix a specific problem.

### C. Mishandled values

**Incorrect conditional check:** We identified 31 bugs in this category. Each of these bugs occur when a conditional check is either incorrect or missing entirely. The incorrect checks cause parts of the code to be skipped entirely or executed when they

[1]We describe bugs using system name-bugID.

should not be. This puts the system in an unexpected state, which causes the system to hang indefinitely.

An example of a missing conditional check is Squid-1968. Here, an internal domain name service (DNS) map used by Squid to keep track of queries can become corrupted, causing Squid to hang occasionally. The fix for this bug is simple and involves the addition of a conditional check to prevent the map corruption.

We next illustrate when an incorrect conditional check can cause a system to hang with Cassandra-5064. In this bug, when altering a table which contains a "collection," a group of items, the system can hang. The fix for this issue is to add a flag to the conditional, forcing the code within the conditional check to be executed under the conditions causing the bug.

**Improper error handling:** We identified 22 bugs in this category. These bugs occur when errors are mishandled by handlers. Many of these bugs occur as the result of multiple distributed components interacting with each other unexpectedly.

An example of improper error handling causing the system to hang is Cassandra-6735. Here, when an exception is thrown while flushing "memtables" during shutdown, the system hangs because the unhandled exception interrupts thread shutdown. The fix for these types of problems is to make sure to handle the error in some way.

These bugs can also occur when a functional error (e.g., Java "IOException") handler is incomplete. For example, in MySQL-11729 when an error causes a system restart, the system hangs as a result of the threads failing to close an open resource. As before, the fix to these bugs are to modify the error handling mechanism to correctly handle the error.

**Unanticipated return value:** We identified 13 bugs in this category. These bugs occur when a function returns a value the developers did not expect.

In Squid-1484, which is an example of the bugs in this category, the conditional check in a `while` loop does not consider that the return value of `open` to `/dev/null` could be -1, ultimately causing the system to hang. The fix for these bugs is to add a new conditional check, modify an existing conditional, or throw an appropriate error (e.g., Java IOException) to handle the unanticipated return value.

**Unanticipated input:** We identified 14 bugs in this category. Each bug in this category is the result of an unanticipated input causing the system to hang. In particular, we found a large portion of these bugs result from an unexpected large input.

An example of these bugs is Apache-45856. This bug causes Apache to hang when trying to open a FastCGI log larger than 2147483647 bytes. The typical fix to problems in this category is to use the appropriate function call to handle the large file.

**Incorrect value definition:** We identified 16 bugs in this category. These bugs occur when a flag or value has been either incorrectly set or is undefined.

An example of the bugs in this category is MySQL-9814. Here, a function fails to reset an error reporting flag, which causes the system to hang. The solution to this bug is typical of other bugs in this category, which is to add the undefined value.

**Observation 2** *Mishandled values were the most common cause for hangs for all the systems we studied. Developing rules to identify these problems is difficult.*

### *D. Concurrency*

**Deadlock:** We identified 24 bugs in this category. These bugs all involve cases where two or more different operations are waiting for each other to complete. As neither operation can make any progress until the other finishes, the system will hang. These situations typically occur when accessing shared system resources (e.g., system log).

An example bug assigned to this category is MySQL-54332. In this bug, one connection locks table $T_1$ while another connection locks table $T_2$ and then executes the INSERT DELAYED command on table $T_1$. As both tables are locked, the system will hang. The built-in deadlock detector of MySQL should detect and prevent this deadlock. However, any deadlocks involving the INSERT DELAYED command are missed by the deadlock detector in this version of MySQL and thus the server hangs. The fix for this bug was to update the deadlock detector in order to ensure the deadlock was correctly detected.

Deadlocks can also occur due to components waiting on thread operations to complete. For example, HDFS-3541 is a deadlock bug occurring as the result of a flag not being set while handling a thread interrupt. The fix is to notify other threads that an interrupt has occurred.

**Race condition:** We identified 7 bugs in this category. Each of the bugs in this category occur when a sequence of operations leads to the system hanging. These bugs typically only manifest when operations occur in only specific orderings, making them difficult to reproduce.

An example of the type of bugs in this category is Hadoop MapReduce-2504. In this bug, a thread is interrupted and then stopped concurrently. This causes the wrong interrupt handler to process the interrupt and incorrectly updates the state of the system. This incorrect state update causes the thread which should have handled the interrupt to hang, waiting for an interrupt which will never occur. These bugs are typically fixed by modifying the application code to ensure the race condition cannot occur. In this case, the bug was fixed by ensuring the interrupt was consumed by the correct interrupt handler using different system states.

**Observation 3** *Despite the fact that concurrency bugs have been well studied, unexpected distributed component interactions can cause systems to hang. Automatically detecting these types of interactions is difficult.*

There are many well known static analysis techniques designed to identify and prevent concurrency bugs ahead of time [15], [14]. However, unexpected component interactions in a distributed environment can lead to deadlocks which existing approaches cannot detect. Identifying a specific concurrency bug does not help detect other bugs in many cases, leading to false negatives.

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

Fig. 3: The patch for Apache-37680. The issue is an endless loop caused by `status` return value not being checked properly.

## IV. Issues with Existing Techniques

Previous work [13] has developed heuristics from careful manual analysis of bug reports which then can be applied to code bases in order to detect new bugs. For example, in Apache, the `setsockopt` function can cause performance problems under certain contexts. A simple rule based checker can look for this function, raising a warning when found. However, simple rule-based checkers cannot be written for all cases. In this section we discuss in detail why simple rule-based checkers will be difficult to develop for many of the bugs we studied.

### A. False Positives

A false positive occurs when a static detector identifies a segment of code as containing a bug when there actually is no bug present. These occur as a result of generic rules being applied to large code bases. To illustrate why these false positives occur, we discuss Apache-37680. This bug was discovered when a user changed some configuration options in the Apache configuration file and did a graceful restart. Instead of having the web server come back online as normal in a few seconds as expected, the user found that the server was hung, consuming 100% CPU in the process.

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

A snippet of the patch to fix this bug is given in Figure 3. As the figure shows, there is only one major change to the application. As shown, instead of having a constant value being passed to the `apr_socket_opt_set` function, a variable, `use_nonblock` is passed to the function. This variable controls whether a blocking or non-blocking call should be made. On a graceful server restart, this variable is set to allow non-blocking calls to be made on the socket and preventing the infinite loop.

An example type of general rule-based static approaches could be developed from this patch would be to check if a constant value is passed to a function as a parameter. The problem with doing this, however, is most of the time, when a constant value is passed to a function, there is no problem. Thus, applying rule-based detectors to these types of bugs is difficult due to the large number of false positives they produce.

### B. False Negatives

A false negative occurs when a static detector fails to identify a segment of code as containing a bug when there actually is a bug present. These can occur when bug are fixed with bug-specific patches. Any rules generated from such bugs would be sensitive to code changes, causing missed detections as context changes. Although the patches themselves may not be very complex, they usually fix a specific issue and are difficult to generalize to other problems.

To illustrate this issue we first discuss Hadoop-MapReduce-3186. In this version of Hadoop, a resource management component called Yet Another Resource Negotiator (YARN) [1] was introduced for improved scalability. Although the resource manager can improve the resource use of applications, it can also cause new problems. In Hadoop-MapReduce-3186, if the global resource manager is restarted while a job is being executed, the job hangs indefinitely in the running state. The cause of the bug is a result of a lack of a timeout being set in conjunction with improper exception handling. As a result, we assign this bug to the mishandled values category.

Because the cause of bug itself is relatively complex, the patch provided by developers is also complex. A snippet of the patch, with comments removed, is shown in Figure 4. The patch involves about 190 lines and changes to functions in four different files.

This type of patch is tailored to fix the specific problem in the bug report and cannot be generalized to help fix other problems. As the code snippet shows, there are very few, if any rules that can be generated from this problem. Generating rules from the statements in the entire patch itself would be difficult. Because code is routinely changed, code additions and deletions may cause the rules to be ineffective, thus causing false negatives.

### C. Observations

We have estimated the false positives and false negatives that would occur if rule-based bug detection techniques were applied to the bugs we studied. Our results are shown in Figure 5. As shown, we found that a significant portion of bugs in each category can generate either false positives, false negatives, or both.

```
...
+    try {
+      response = makeRemoteRequest();
+      retrystartTime = System.currentTimeMillis();
+    } catch (Exception e) {
+      if (System.currentTimeMillis() -
 retrystartTime >= retryInterval) {
+        eventHandler.handle(new JobEvent(this.
 getJob().getID(),
+
 JobEventType.INTERNAL_ERROR));
+        throw new YarnException("Could not contact
 RM after " +
+                                retryInterval + "
 milliseconds.");
+      }
+      throw e;
+    }
+    if (response.getReboot())
...
```

Fig. 4: A snippet of the developer patch for Hadoop-MapReduce-3186 which was assigned to the mishandled values category. This patch is large and complex, making false negatives an issue for any rules generated from such a patch.

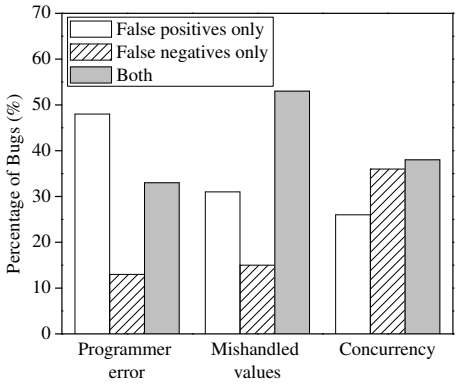


Fig. 5: The distribution of bugs causing only false positives, only false negatives, and both false positives and false negatives.

## V. Conclusions and Future Work

In this paper, we have presented a characteristic study of 177 real world software hang bugs. Our study shows that the root causes of those hang bugs vary from system to system with incorrect conditional check, incorrect timeout setting, and deadlock being the top three root causes. We examine the applicability of rule-based bug detection techniques to the software hang bugs we studied and find that they fall short in terms of high false positives and high false negatives. Our bug study has indicated that existing static analysis techniques are not sufficient for detecting and diagnosing server hang bugs. In order to address the issues of false positives and false negatives, we are exploring ways of combining runtime hints with static analysis techniques to achieve precise root cause analysis. We believe that the observations of our study enable the development of new tools and make automated server hang bug diagnosis a feasible reality in the near future.

## Acknowledgment

We thank the anonymous reviewers for their valuable comments. This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, NSF CAREER Award CNS1149445, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, IBM Faculty Awards and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, ARO, or U.S. Government.

## References

[1] Apache Hadoop 2.6.0-YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[2] Amazon web services. http://aws.amazon.com/.

[3] Apache. http://httpd.apache.org/.

[4] Apache Cassandra. http://cassandra.apache.org/.

[5] Apache Tomcat. http://tomcat.apache.org/.

[6] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *Software, IEEE*, 2008.

[7] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *EuroSys*, 2010.

[8] D. Dean, H. Nguyen, and X. Gu. UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *ICAC*, 2012.

[9] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *SoCC*. ACM, 2014.

[10] D. J. Dean, H. Nguyen, P. Wang, and X. Gu. Perfcompass: toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds. In *HotCloud*, 2014.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[12] Hadoop Distributed File System. http://hortonworks.com/hadoop/hdfs/.

[13] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, 2012.

[14] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[15] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI*, 2012.

[16] Lighttpd. http://www.lighttpd.net/.

[17] Y. Lin and S. Kulkarni. Automatic repair for multi-threaded program with deadlock/livelock using maximum satisfiability. In *ISSTA*, 2014.

[18] MySQL. http://www.mysql.com/.

[19] Amazon Web Services suffers outage, takes down Vine, Instagram, others with it. http://www.zdnet.com/article/amazon-web-services-suffers-outage-takes-down-vine-instagram-others-with-it/.

[20] Squid-cache. http://www.squid-cache.org/.

[21] G. Upadhyaya, S. P. Midkiff, and V. S. Pai. Automatic atomic region identification in shared memory spmd programs. In *OOPSLA*, 2010.

[22] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[23] X. Wang, Z. Guo, X. Liu, Z. Xu, H. Lin, X. Wang, and Z. Zhang. Hang analysis: Fighting responsiveness bugs. In *Eurosys '08*, 2008.

[24] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, 2008.

[25] D. Weeratunge, X. Zhang, and S. Jaganathan. Accentuating the positive: atomicity inference and enforcement using correct executions. In *OOPSLA*, 2011.

[26] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *OSDI*, 2010.

[27] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*, 2014.