HangFix: Automatically Fixing Software Hang Bugs for Production Cloud Systems

Jingzhu He
North Carolina State University
jhe16@ncsu.edu

Ting Dai
IBM Research
ting.dai@ibm.com

Xiaohui Gu
North Carolina State University
xgu@ncsu.edu

Guoliang Jin
North Carolina State University
guoliang_jin@ncsu.edu

ABSTRACT

Software hang bugs are notoriously difficult to debug, which often cause serious service outages in cloud systems. In this paper, we present HangFix, a software hang bug fixing framework which can automatically fix a hang bug that is triggered and detected in production cloud environments. HangFix first leverages stack trace analysis to localize the hang function and then performs root cause pattern matching to classify hang bugs into different types based on likely root causes. Next, HangFix generates effective code patches based on the identified root cause patterns. We have implemented a prototype of HangFix and evaluated the system on 42 real-world software hang bugs in 10 commonly used cloud server applications. Our results show that HangFix can successfully fix 40 out of 42 hang bugs in seconds.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Availability**; **Cloud computing**; • **Software and its engineering** → **Software performance**; **Software testing and debugging**.

KEYWORDS

System reliability, Automatic bug fixing, Hang bug characterization

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421288

ACM Reference Format:
Jingzhu He, Ting Dai, Xiaohui Gu, and Guoliang Jin. 2020. HangFix: Automatically Fixing Software Hang Bugs for Production Cloud Systems. In *Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3419111.3421288

1 INTRODUCTION

Many production server systems (e.g., Cassandra [6], HBase [2], Hadoop [7]) are migrated into cloud environments for lower upfront costs. However, when a production software bug is triggered in cloud environments, it is often difficult to diagnose and fix due to the lack of debugging information. Particularly, software hang bugs causing unresponsive or frozen systems instead of system crashing are extremely challenging to fix, which often cause prolonged service outages. For example, in 2015, Amazon DynamoDB experienced a five-hour service outage [3, 4] affecting many AWS customers including Netflix, Airbnb, and IMDb. The root cause of the service outage was a software hang bug where an improper error handling kept sending new requests to the overloaded metadata server, causing further cascading failures and retries. In 2017, British Airways experienced a serious service outage with a penalty of more than £100 million [5] due to a software hang bug triggered by corrupted data during data center failover.

Unfortunately, software hang bugs are notoriously difficult to debug because they typically produce little diagnostic information. Recent studies [13–15, 20, 21] have also shown that many hang bugs are caused by *unexpected* runtime data corruptions or inter-process communication failures, which makes those hang bugs particularly difficult to be caught during the testing phase. Although previous bug detection tools [12, 14, 21, 29, 38] can detect those hang bugs, production service outage cannot be truly resolved until the hang bugs are correctly fixed. Otherwise, the service outage will happen again when the bug-triggering condition is met again in the production system.

```
//FSHDFSUtils.java                HBase-8389(v0.94.3)
    long recoveryTimeout = conf.getInt("hbase.lease.
         recovery.timeout", 900000) + startWaiting;
 48 public void recoverFileLease(..., final Path p,
 49                             ...) throws IOException {
     ...
 62  boolean recovered = false;
 63  int nbAttempt = 0;
 64- while (!recovered) {
   + for (int nbAttempt = 0; !recovered; nbAttempt++){
 65-  nbAttempt++;
      ...
 71   recovered = dfs.recoverLease(p);//send to HDFS
   +  if (recovered) break;
   +  if (checkIfTimedout(conf, recoveryTimeout,
   +                     nbAttempt, p, startWaiting))
   +     break;
      ...
104 }}

   +boolean checkIfTimedout(final Configuration conf,
   + final long recoveryTimeout, final int nbAttempt,
   + final Path p, final long startWaiting) {
   + if (recoveryTimeout <
   +    EnvironmentEdgeManager.currentTimeMillis()) {
   +   LOG.warn(...);
   +   return true;
   + }
   + return false;
   +}
```

Figure 1: When the blocks are corrupted, the `recoverLease()` function keeps polling the recovery results, getting "false" and sending a new recovery request, hanging in an infinite loop. "+" means added code, representing the manual patch for this bug.

Previous work [27, 30, 38] often assumes full stack domain knowledge or complete source code (e.g., distributed platform and application), which however is often impractical for the production cloud system. Cloud environments often involve multiple parties where infrastructure, distributed platforms (e.g., MapReduce, Cassandra and HDFS), and applications are developed by different parties. Therefore, it is essential to design *domain-agnostic*, *byte-code-based* software hang bug fixing techniques that can be applied to different cloud systems without requiring domain knowledge or source code.

1.1 A Motivating Example

We use the HBase-8389 bug as an example to describe how the hang bug is triggered and how it causes cloud service outage. Figure 1 shows the buggy code snippet and the patch. HBase inquires the previous recovery results of the path p from HDFS at line #71. Due to an unexpected data corruption, the return results from all the inquiries are always false, which causes HBase to get stuck in this infinite loop between lines #64-104. This bug is difficult to fix because HBase does not produce any log information and HDFS provides many misleading error messages. It took the developer 24 days to fix the bug after submitting 10 versions of unsuccessful patches. The final patch created by the developer is actually quite simple, that is, adding a timeout check mechanism to break out of the loop when the recovery operation persistently fails after a certain number of retries.

1.2 Our Contribution

This paper presents HangFix, a *domain-agnostic*, *byte-code-based* software hang bug fixing tool that aims at providing automatic bug fix patches for software hang bugs that are triggered in production cloud computing environments. We currently focus on fixing Java hang bugs in commonly used cloud server systems. Upon a hang bug detected by a bug detection tool [12, 14, 21, 29, 38] in the production environment, HangFix takes the application byte code and hang bug triggering test cases as inputs, and produces a hang bug fix patch as the output. To make HangFix practical for production cloud environments, we do not require any application source code or application-specific knowledge.

In order to create effective and non-intrusive fixes [31], HangFix consists of four closely integrated steps shown by Figure 2: 1) *hang function localization* which leverages stack trace analysis to pinpoint exact function which causes the application to get stuck in the hang state; 2) *likely root cause pattern matching* which leverages static code analysis over the identified hang function to automatically match a set of common hang bug root cause patterns; 3) *patch generation* which automatically produces a hang bug fix patch based on the matched likely root cause pattern by inserting existing exceptions or timeout mechanisms to break the application out of the stuck state; and 4) *patch validation* which applies the initial bug detection tool and the hang function detection to the patched code to validate whether the hang bug still exists. We also test the patched code with the whole regression test suites to check whether our fix violates any required regression tests. The patch is deemed to be successful only if the patched code passes both bug detection and regression tests.

Specifically, this paper makes the following contributions:

- We present a new hang bug fixing framework to automatically fix a hang bug that is triggered in production cloud environments.
- We describe a hang bug root cause pattern matching scheme that can quickly classify hang bugs into different likely root cause types.
- We develop an automatic hang fix patch generation system that can produce proper patched code according to identified likely root cause patterns.
- We conduct an empirical study over 237 real production hang bugs to quantify the generality of our root cause patterns.

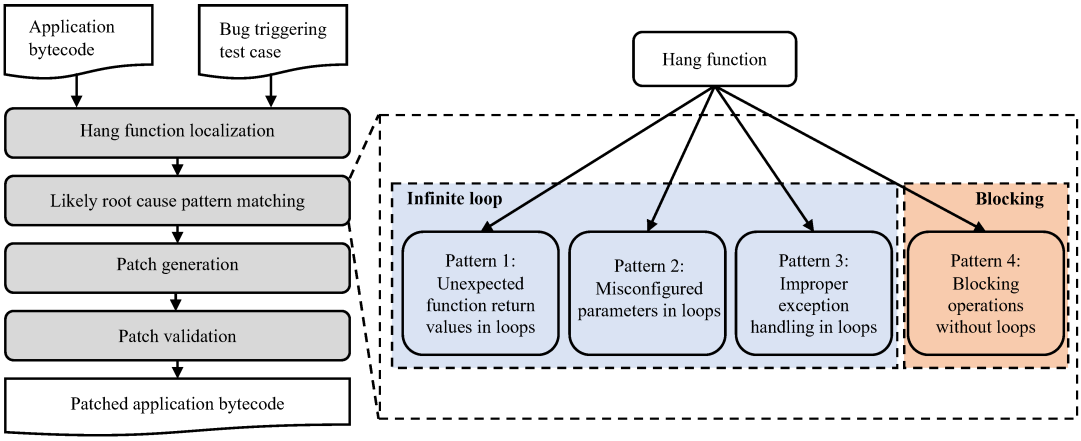


Figure 2: The system architecture of HangFix.

We have implemented a prototype of HangFix and evaluated our system using 42 reproduced real-world software hang bugs in 10 commonly used cloud server systems (e.g., Cassandra, HDFS, Mapreduce, HBase). HangFix successfully fixes 40 of them in seconds, many of which take days for the developers to manually fix them. Our empirical bug study shows that our likely root cause patterns cover 76% of 237 hang bugs. HangFix can correct the hang symptom for all of those matched bugs and completely fix the root cause for 75% of them.

The rest of the paper is organized as follows. Section 2 describes the hang function localization scheme. Section 3 describes the root cause pattern matching and patch generation schemes. Section 4 presents the empirical bug study and experimental evaluation results. Section 5 compares our work with related work. Finally, the paper concludes in Section 6.

2 HANG FUNCTION LOCALIZATION

After a hang bug is detected in production cloud environments, HangFix leverages stack traces to localize the hang function. Since the hang function often repeatedly appears in the stack trace, we can capture the abnormal behavior of the hang function after the hang bug is triggered. Specifically, HangFix uses the `jstack` [1] tracing tool to perform continuous trace dumps after a hang bug is reported by exising hang bug detection tools (e.g., [14, 21]). We then analyze the stack trace to localize the hang function. Intuitively, the hang function will repeatedly appear in the stack trace as the application's execution gets stuck in the hang function.

Figure 3 shows the dumped stack traces of `main` thread when Compress-451 bug is triggered. The trace contains the running status of each thread's status and the invoked Java functions with the invoking line number. The trace also indicates the call stack of the invoked functions in each thread.

```
//Dump 1
"main" #1 prio=5 os_prio=0 tid=0x00007f899c00b000 nid=0
    x76b9 runnable [0x00007f89a27fa000]
 java.lang.Thread.State: RUNNABLE
 at java.io.FileInputStream.readBytes(Native Method)
 at java.io.FileInputStream.read(FileInputStream.java
    :233)
 at org.apache.commons.compress.utils //hang function
    .IOUtils.copy(IOUtils.java:47)
 at testcode.testCopy(testcode.java:32)
 at testcode.main(testcode.java:12)

//Dump 2
"main" #1 prio=5 os_prio=0 tid=0x00007f899c00b000 nid=0
    x76b9 runnable [0x00007f89a27fa000]
 java.lang.Thread.State: RUNNABLE
 at java.io.FileOutputStream.writeBytes(Native Method)
 at java.io.FileOutputStream.write(FileOutputStream.java
    :326)
 at org.apache.commons.compress.utils //hang function
    .IOUtils.copy(IOUtils.java:49)
 at testcode.testCopy(testcode.java:32)
 at testcode.main(testcode.java:12)

...
```

Figure 3: Continuously dumped stack traces of `main` thread for the Compress-451 hang bug.

The `testcode.java` is the bug triggering test case. The stack trace shows that two functions of `testcode.java` are invoked, i.e., `main()` and `testCopy()`. The `testcode.testCopy()` function invokes the `compress.utils.IOUtils.copy()` function in the Compress system.

To localize the hang function, we extract the repeated function(s) by comparing the function names among different stack trace dump files. In each thread, we extract the repeated function at the top of the call stack as the hang function. For example, in the Compress-451 bug stack trace shown in Figure 3, HangFix extracts three repeatedly invoked functions: the `testcode.main()`, `testcode.testCopy()` and `compress.utils.IOUtils.copy()` functions. The top of the call stack, that is, the last function on the call

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

114 protected void drain(InputStream dis, long bytesRead)
      throws IOException {
115   long toSkip = totalSize() - bytesRead;
116   toSkip = toSkip - dis.skip(toSkip);
117   while (toSkip > 0) {
118-   toSkip = toSkip - dis.skip(toSkip);
    +   long skipped = dis.skip(toSkip);
    +   toSkip = toSkip - skipped;
    +   if (skipped <= 0){ //immediate termination
    +    throw new IOException("Unexpected return"
    +        +"value causes the loop stride to be"
    +        +"incorrectly updated.");}
119   }
120 }
```

Figure 4: Example of hang bug pattern #1 and its fixing strategy. When `InputStream dis` is inaccessible or corrupted by bad encoding `dis.skip` can return -1 or 0, and -1/0 is used as the stride. "⟶" represents the function call invocation. "-" means deleted code and "+" means added code, representing the patch generated by HangFix.

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

Figure 5: When the blocks are corrupted, the `recoverLease()` function keeps polling the recovery results, getting "false" and sending a new recovery request, hanging in an infinite loop. "+" means added code, representing HangFix's patch for this bug.

path, is identified as the hang function. For example, in the Compress-451 bug, the `compress.utils.IOUtils.copy()` function is extracted as the hang function since it repeatedly appears and is the last hop on the call path. The `compress.utils.IOUtils.copy()` function invokes different statements (i.e., `FileInputStream.read()` and `FileOutputStream.write()`) inside the loop at two different dumps.

3 ROOT CAUSE PATTERN MATCHING AND PATCH GENERATION

In this section, we describe our pattern-driven approach to quickly identifying likely root causes of hang bugs and automatically generating hang fix patches. HangFix performs root cause pattern matching by analyzing intermediate representation (IR) code produced by Java Soot compiler [9]. The patched code is created by inserting proper IR code into the hang function.

3.1 Likely Root Cause Pattern #1: Unexpected Function Return Values in Loops

Root cause pattern description: If the hang function involves loops and the loop stride depends on the return values of some functions which could be either Java library functions (e.g., `InputStream.skip`) or application-specific functions (e.g., `dfs.recoverLease(p)`), we infer that the hang is caused by unexpected function return values. For example, the HBase-8389 bug shown by Figure 5 falls in this root cause pattern. For this pattern, HangFix extracts the loop index, stride, and bound abstraction as "`while(index == 0){stride = Method(); index += stride;}`". The loop stride depends on an application-specific RPC invocation `dfs.recoverLease(p)`, matching this likely root cause pattern.

Patch generation: If the *loop stride impacting* function f_i is a known Java library function, our patch is to add proper checks over possible incorrect return values and break out of the hang state by throwing an exception that has been declared in the hang function in order to minimize unwanted side-effect from our patch. If function f_i does not contain "throws exception" clause in its signature, HangFix checks the call stack of f_i backwards until it identifies the n-hop caller function of f who declares a checkable exception in its function signature. HangFix then inserts the same checkable exception in the signatures of the function f_i and its i-hop callers, $i = 1, 2, ..., n-1$. If there are more than one checkable exceptions, HangFix chooses the first and most specific one.

We choose to fix this type of hang bug through immediate loop termination by throwing exceptions because we observe that Java library functions typically produce persistent errors. For example, the `InputSteam.skip(long len)` function can return any value in the following three subsets:

$\{-1\} \cup \{0\} \cup [1, |n|]$. However, once it returns -1 (i.e., end of file) or 0 (i.e., corruption), it cannot return any positive values in the subsequent invocations.

If the loop stride impacting function f_i is an application-specific function (e.g., RPC calls to a remote server), our fix is to insert a timeout checking code at the tail of the loop to break out of the loop if the loop execution time exceeds a certain timeout threshold. In contrast to Java library functions which typically have persistent return errors, the errors from application-specific functions can be transient (e.g., file reading errors due to transient network failures). Therefore, instead of terminating the loop right away, HangFix allows the application to retry the function invocation within a certain time limit but prevents the application from entering a hang state by enforcing the timeout check. Specifically, the timeout checking code consists of an `if` branch with the condition of $var_e >= var_{new}$. The var_e denotes the elapsed time of the loop's execution while var_{new} is a pre-set timeout variable. Inside the `if` branch, a known exception is thrown with a timeout error message. We will discuss how HangFix picks the right timeout value in Section 3.4.

Examples: Figure 4 shows the patch produced by HangFix for the Cassandra-7330 bug. The loop stride impacting function is the Java library function `InputStream.skip()`. HangFix inserts a check after the invocation of `skip()`, which includes introducing a local variable `skipped` to store the return value, generating an extracted error value set[1] (i.e., <= 0), comparing `skipped` with "<= 0", and throwing an `IOException` which is already defined in the hang function with an error message.

Figure 5 shows the patch produced by HangFix for the HBase-8389 bug. The loop stride impacting function is the application-specific function `dfs.recoverLease()`. The patch adds a timeout checking code block at the end of the `while` loop body, which is similar to the manual patch that is produced by the developer shown by Figure 1.

3.2 Likely Root Cause Pattern #2: Misconfigured Parameters in Loops

Root cause pattern description: If the hang function involves loops and the loop stride depends on a configurable, constant variable, we infer the hang is caused by some misconfigured or incorrectly hard-coded values. For example, the Hadoop-15415 bug shown by Figure 6 falls in this root cause pattern where the `bufferSize` parameter is misconfigured to be 0 at line #97 and passed in as an argument at line #74. `InputStream in` performs the `read()` operation

[1]HangFix extracts the error value set by analyzing the loop body to infer the error-inducing loop stride values (e.g., <=0) and possible return values of the Java library functions (e.g., $\{-1\} \cup \{0\} \cup [1, |n|]$).

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

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

74 public static void copyBytes(InputStream in, ..., int
       buffSize) throws IOException {
  +  if(buffSize == 0)       //early termination
  +    throw IOException("buffSize cannot be 0");
     ...
 77   byte buf[] = new byte[buffSize];
 78   int bytesRead = in.read(buf);
 79   while (bytesRead >= 0) {
     ...
 84    bytesRead = in.read(buf);
 85 }}
```

Figure 6: Example of hang bug pattern #2 and its fixing strategy. Misconfiguration causes `bufferSize` to be 0, which in turn makes the `InputStream in` perform `read` operation on a zero-size byte array and return 0. "⟶" represents the function call invocation, while "⇢" represents the data dependency flow. "+" means added code, representing the patch generated by HangFix.

on a zero-size byte array and returns zero at line #84, indicating nothing being read from the continuous flow and the end of the flow can never been reached (i.e., `bytesRead < 0`). As a result, the `copyBytes()` function endlessly spins in the loop. HangFix identifies this bug as a root cause pattern #2 bug because the hang function `copyBytes()` at line #74 contains a loop and the loop stride is constantly updated by a configurable parameter `buffSize` in each loop iteration. Specifically, HangFix first retrieves the loop index, stride and bound abstraction for the `InputStream.read(byte[] buf)` function as "`while(index < bound){index += buf.size;}`", where `bound` is the end of the `InputStream`. After analyzing the data dependency flow, HangFix identifies that the stride is misconfigured at line #97-98, causing `bytesRead` to be constantly assigned with zero. Different from Pattern #1, here 0 is a legitimate return value for the `in.read()` function. So the root cause is the misconfigured parameter.

Patch generation: To fix Pattern #2 hang bugs, we add proper checkers over the misconfigured values or arbitrarily hard-coded values before they are used in the loop. HangFix cuts those error values' data flow by throwing a known exception before the fault gets propagated and manifested as

system hang, which we call *early termination*. Early termination saves unnecessary resource consumption compared with immediate or delayed termination.

During pattern identification, HangFix parses the hang function's call graph and the error-prone variable's data dependency flow to locate the misconfiguration (or faulty assignment) statement s_1. If the error-prone value is not checked properly, HangFix inserts a checker (i.e., c_1) after s_1. The checker is an `if` branch with the condition of $v == v_{err}$. If v has more than one error values, HangFix generates a combined condition in the form of $v >= v_{err_{min}}$ or $v <= v_{err_{max}}$. Inside the `if` branch, a known exception is thrown with an error message such as "variable v is misconfigured/-falsely assigned with v_{err}, affecting loop stride, leading to an infinite loop."

In addition, HangFix inserts another checker (i.e., c_2) at the beginning of the hang function before the loop is executed, if 1) the statement s_1 cannot be identified or 2) the error-prone variable is a hang function's parameter and the hang function is "public", meaning it can be directly accessed by user-defined classes or classes in other integrated systems. This c_2 checker is similar as the c_1 checker but with a different error message, e.g., "variable v cannot be v_{err}, which can lead to an infinite loop."

Examples: The Hadoop-15415 bug in Figure 6 can be fixed using this patching strategy. HangFix's pattern identification indicates that the infinite loop happens when the `buffSize` is non-positive. HangFix also detects that `buffSize` is non-negative because it is used as an array's size at line #77. Intersecting "non-positive" with "non-negative," HangFix generates the error-prone value as $\{0\}$. After detecting that `buffSize` is passed in as a parameter and configured at line #97-98, HangFix inserts a checker after line #98, including an `if` branch with condition "`buffSize == 0`" and an `IOException` with error message "misconfigured `buffSize` with 0." Since the `copyByte()` function at line #74 is public, it can be accessed by user-defined classes or classes in any other integrated systems (e.g., HBase, Hive, etc). The argument value of `buffSize` passed in from those classes are inaccessible in our analysis currently. Thus, another checker inside the `copyByte()` function before the loop is necessary. After line #74, HangFix inserts an `if` branch with condition "`buffSize == 0`" and an `IOException` with error message as "`buffSize` cannot be 0."

3.3 Likely Root Cause Pattern #3: Improper Exception Handling in Loops

Root cause pattern description: If the hang function involves loops and the loop stride update is skipped due to some exceptions, we infer the hang is caused by improper exception handling in loops. For example, the Cassandra-9881 bug shown by Figure 7 falls in this root cause pattern. When `dataFile` is corrupted, an `IOException` is raised by the `readWithShortLength()` function at line #130. The `IOException` gives up the correct execution of `readWithShortLength()` and skips the `readLong()` function at line #134. The above read functions are both loop index-forwarding operations. This `IOException` is then simply ignored at line #140. Without moving the loop index forwards at line #130 and #134, the `scrub()` function keeps reading from the location, spinning forever. HangFix identifies this bug as a Pattern #3 bug, because in the exception handling control flow path `120→129→130→139→140→141`, there is no loop stride update along the path. Specifically, HangFix first extracts all the invocations of the `DataInput` instance `dataFile`, including `isEOF()` at line #120, `readUnsignedShort()` and `readByte()`[2] at line #130, and `readLong()` at line #134. HangFix then generates the loop index, stride, and bound abstraction for the above `DataInput` functions as "`while(index < bound){index += 2; index += 1; index += 8;}`".

```
//CompactionManager.java          Cassandra-9881(v2.0.8)
436  private void scrubOne(...) throws IOException {
       ...
444    scrubber.scrub();
       ...
459  }

//Scrubber.java
103- public void scrub(){
   + public void scrub() throws IOException {
       ...
120    while (!dataFile.isEOF()){
         ...
   +     int index = 0;
129      try{                  /*dataFile is corrupted*/
130        key = sstable.partitioner.decorateKey(
     ByteBufferUtil.readWithShortLength(dataFile));
   +       index += 3;//trace index change
           ...
134        dataSize = dataFile.readLong();
   +       index += 8;//trace index change
           ...
139      } catch (Throwable th){
140        ...; //ignore Exception
   +       if(index == 0) //no index update,
   +         throw th;    //immediate termination
141      }
         ...
       }}
```

Throw IOException

Figure 7: Example of hang bug pattern #3 and its fixing strategy. Data corruption causes `readWithShortLength()` to throw exception at line #130-131, which makes the loop skip the index updating statement (i.e., zero-stride) at line #134. "→" represents the function call invocation, while "- - >" represents the control flow. "-" means deleted code and "+" means added code, representing the patch generated by HangFix.

[2] `readUnsignedShort()` and `readByte()` are the callees of `readWithShortLength()`. To save space, we omit them in Figure 7.

Moreover, all the abstractions for the loop stride (i.e., `index += 2` and `index += 1` at line #130, `index += 8` at line #134) do not appear in the exception handling control flow, matching this root cause pattern.

Patch generation: To fix this type of hang bugs, HangFix conducts *index tracing* to check whether the loop has no stride or ineffective loop stride. If the index updating operation is not correctly executed (i.e., the loop index does not change) at one iteration, this loop does not contain effective stride along this iteration's control flow. HangFix inserts a counter variable `index` with the value of `v` at the beginning of a loop. It then extracts the loop index and bound abstraction (i.e., `index += stride`) for each index updating operation, and inserts such abstraction after the operation.

If the loop index is not updated in the exception or error handling control flow, HangFix first tries to fix the hang bug by re-executing the loop index updating operations to restore loop stride properly. Specifically, HangFix inserts a checker inside the exception handling block (e.g., `catch` block) or after the error return statement. The checker is an `if` branch with condition of "`index == v`", inside which, HangFix inserts the unexecuted index-updating operations. If the loop has multiple index-updating operations, HangFix chooses the first one in each control flow from the loop header to the exception/error handling block. The chosen operation is inserted with the conditions along its control flow. For example, for the following control flow `if(cond){op1();op2();} else{op3();}`, HangFix inserts "`if(index == v){if(cond){op1();} else{op3();}`" in the checker. If the loop stride restoration fails to fix the hang bug, HangFix terminates the loop with a known exception to break out of the hang state.

Examples: Figure 7 shows the patch produced by HangFix for the Cassandra-9881 bug. HangFix starts index tracing by inserting a counter variable `index` with the original value 0 before the `try` block at line #129. HangFix first identifies that the `readWithShortLength()` function at line #130 and the `dataFile.readLong()` function at line #134 are index updating operations and can be abstracted as "index += 3" and "index += 8", respectively. HangFix then inserts these abstractions after line #130 and #134. In the exception handling block at line #139-141, HangFix inserts an `if` branch with the condition of `index == 0` and a `throw` exception statement to immediately terminate the loop. This is doable because HangFix adds the `throws IOException` clause in the `scrub()` function's signature at line #103 and relies on Cassandra's existing exception handling mechanisms to repair failures by propagating the exception backwards to the caller function, `scrubOne()`, and we refer to it as *exception heritage*. HangFix does not re-execute the index updating operation inside the `catch` block because re-executing the

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

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

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

Figure 8: Example of hang bug pattern #4 and its fixing strategy. `Inflater.inflate()` is a blocking-pone function. When an ORC file is corrupted, conducting the `inflate()` operation on a corrupted file causes an infinite loop in the underlying JNI code. "⟶" represents the function call invocation. "-" means deleted code and "+" means added code, representing the patch generated by HangFix.

operation (i.e., `readWithShortLength`) will still throw exceptions due to the corrupted `dataFile` and Cassandra will still hang.

3.4 Likely Root Cause Pattern #4: Blocking Operations Without Loops

Root cause pattern description: If the hang function does not contain any loop and the hang function consists of some blocking operation (e.g., Java library functions, JNI methods), HangFix infers that the hang is caused by blocking operations. For example, the Hive-5235 bug shown in Figure 8 falls into this root cause pattern. The hang bug is triggered when the hang function `decompress` invokes a blocking Java library function `inflate()`. The underlying JNI

code of the library function hangs in an infinite loop when it is invoked over a corrupted ORC file. HangFix classifies this bug as a Pattern #4 bug since the hang function `decompress()` does not contain any loop but one blocking Java library function call.

Patch generation: To fix hang bugs caused by blocking operations, HangFix first isolates the blocking operation from the main application execution using a callable or runnable thread. Next, HangFix adds a timeout check to end the blocking operation after a certain waiting period defined by var_{new} (e.g., `future.get(`var_{new}`)`, `thread.join(`var_{new}`)`. The newly introduced timeout variable var_{new} is first configured with the default value of a known timeout variable v. HangFix extracts the default value by searching the system's configuration files using keywords, such as "timeout," "interval," "block," and "poll." HangFix chooses the variable which matches the most keywords, and assigns its default value v to var_{new}. The rationale is that variables share similar names most likely have similar purposes, thus similar default values. During the patch validation phase, we can adjust the timeout variable values if the patch does not pass any test. We can also leverage timeout value prediction techniques (e.g., [22]) to infer the timeout values more efficiently.

Examples: Figure 8 shows the patch produced by HangFix for fixing the Hive-5235 bug. HangFix first introduces the `timeout` variable with the default value of 5000 milliseconds. This default value is read from the `HiveConf.HIVE_SERVER2_LONG_POLLING_TIMEOUT` variable. HangFix replaces the blocking operation `Inflater.inflate()` at line #94 with a new function called `inflateWithTO()` isolated by a callable thread. The timeout setting is in the `future.get()` function with the timeout variable `timeout`. To break out of the blocking state, the function uses a known exception type `DataFormatException` which has been used by this hang function with a useful log message "endless blocking." So the patch provided by HangFix can not only prevent the service outage caused by the hang bug but also provide useful information for the developer to know the root cause of the auto-patched hang bug.

3.5 Discussion

This paper focuses on hang bug fixing, rather than hang bug detection. We rely on previous work [14, 21] to detect hang bugs. It is possible that hang bug detection tools raise false alarms, which is not the focus of this paper.

We pick those four likely root cause patterns to implement in HangFix based on previous hang bug study results [13–15, 20, 21] as well as our past experiences. However, the root causes of hang bugs are definitely not limited to those patterns only. We have conducted an empirical hang bug study to understand the representativeness of our root cause patterns in real production hang bugs. Our empirical study shows that HangFix root cause patterns can cover all of the hang bugs which are not related to synchronizations. We will describe our hang bug root cause study results in detail in Section 4.

As mentioned in the introduction, we validate the patch produced by HangFix using existing bug detection tools, our hang function localization tool, and the application's regression test suites. Due to the complexity of cloud computing environments and modern server systems, HangFix currently does not provide any theoretical proof on the correctness and completeness of the auto-patched code. Although HangFix proactively takes cautious steps (e.g., reuse existing exceptions) to avoid unwanted effects, HangFix cannot guarantee the automatically generated patches do not bring any side effect to the application especially when the hang function involves application-specific stateful operations. One key objective of our empirical bug study described in Section 4 is to understand the coverage of our root cause patterns for real world production hang bugs.

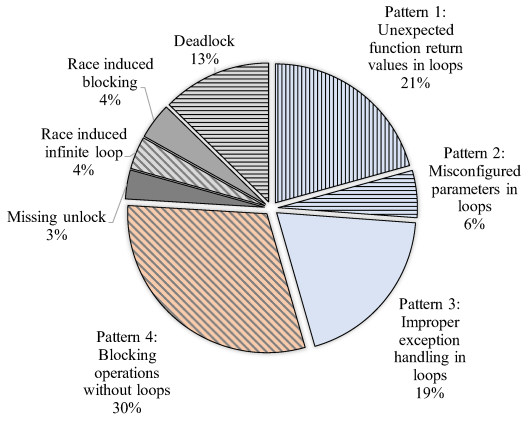
4 EVALUATION

In this section, we present our evaluation results on HangFix. Our evaluation consists of two parts: 1) an empirical study over 237 real hang bugs; and 2) an experimental evaluation over 42 real hang bugs that can be reproduced by us successfully. We first describe our evaluation methodology followed by detailed results and analysis.

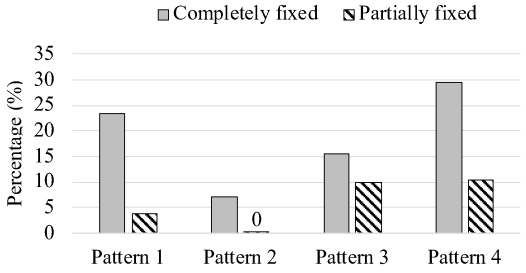
4.1 Evaluation Methodology

Cloud systems: We collect hang bugs from the bug-tracking systems, e.g., Apache Jira [8], for 10 open source cloud systems: Cassandra key-value store, Compress I/O compression library, Hadoop common library, Hadoop Mapreduce big-data processing framework, Hadoop HDFS file system, Hadoop Yarn resource management service, HBase database management system, Hive data warehouse, Kafka streaming platform, and Lucene text searching engine. These 10 systems are representatives of popular open source production systems used in cloud environments. They cover a wide range of different systems, varying from distributed big data processing to log search.

Benchmarks: We use the "hang", "stuck" and "block" keywords to search for hang bugs. However, those keywords can appear in bug descriptions, test cases (the bug causes test cases hanging), or patch evaluation (the proposed patch causes system hanging). Therefore, we manually examine each bug to determine whether it is a real hang bug triggered in production environments. We include both fixed and open bugs, but eliminate bugs labeled as "Not a problem" or "Will not fix." We target bugs occurred in production



(a) Root cause pattern distribution.



(b) Percentage of completely and partially fixed bugs.

Figure 9: The pattern matching and fixing results of the 237 hang bugs in our empirical study. 76% of them fall into HangFix's four root cause patterns. 75% of them can be completely fixed.

systems and eliminate the bugs in the test suites. To the best of our efforts, we collect 237 bugs in total from the 10 commonly used server systems.

Root cause pattern matching and patching evaluation in the empirical study: After collecting these real-world software hang bugs, we manually study each of them to match it with HangFix's likely root cause patterns. Based on our understanding of the root cause of the bug, we analyze whether HangFix's patch can fix the bug completely without introducing new bugs, i.e., we check whether HangFix can fix the bug and does not influence the code logic after jumping out of the infinite loop or blocking. If a manual patch exists for a bug, we compare the manual patch with HangFix's patch. If the bug is not fixed yet or the manual patch is different from HangFix's patch, we analyze the root cause and the patch of the bug to check whether HangFix's patch introduces new bugs or side effects.

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

Implementation: HangFix is implemented on top of Soot compiler [9] in Java, using `BodyTransformer` and `SceneTransformer` to conduct intra- and inter-procedural analysis, and `ForwardFlowAnalysis` to conduct data flow dependency analysis.

4.2 Empirical Study Results

Figure 9a shows the results of pattern matching results of the 237 studied bugs. As shown in the figure, 76% bugs, i.e., a total of 180, fall into HangFix's four patterns. The other 24% bugs are related to synchronization and concurrency.

4.2.1 Negative Case Study. Figure 9b shows the fixing results of the 180 bugs which fall into HangFix's four fixable patterns. 136 out of the 180 bugs can be fixed by HangFix completely. For the 44 bugs partially fixed by HangFix, their manual patches contain application-specific operations or it is required to restore system's state to fix the bugs.

We describe three bug examples, which cannot be fixed by HangFix. We analyze the root cause of the bugs and explain why HangFix's patch is not complete.

Yarn-3999 (Pattern 1): The bug occurs when Yarn system is draining events to an external system, e.g., Zookeeper. When the external system becomes very slow or unresponsive, the Resource Manager hangs on waiting to flush all the events into the external system. When all the events are flushed, the Resource Manager transitions to the STANDBY state. Due to the hang on draining events, the Resource Manager's transition to the STANDBY state cannot be completed, expiring all current applications on the Yarn system. HangFix identifies the bug as Pattern 1 because the bug hangs on an infinite loop and the loop stride depends on a function's invocation. HangFix's patch can ensure the Yarn system will jump out of the draining events loop by adding a timeout, even if not all the events are flushed. However, the Resource Manager still cannot enter STANDBY state. In this case, the Resource Manager cannot work after applying HangFix's patch.

Kafka-1238 (Pattern 3): The bug occurs when a client updates metadata of a Kafka cluster. The Kafka cluster updates the cluster after the metadata is received from the client each time. However, if none of the nodes are alive, the Kafka cluster reports an error and the client retries the updating request endlessly in an infinite loop. The reason is that at least one node needs to be alive to update the whole Kafka cluster. HangFix identifies the bug as the Pattern 3 because loop stride update is skipped. HangFix's patch enables the client to jump out of the requesting infinite loop. However, the Kafka cluster has no live node and the whole

Table 1: 42 reproducible hang bug benchmarks. Although some bugs have the same description, they happen in different functions or classes.

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

Kafka cluster cannot work after applying HangFix's patch. In comparison, the manual patch ensures that there is always at least one live node in the whole Kafka cluster.

HBase-8729 (Pattern 4): This bug occurs when multiple SSH handlers are replaying logs. When the assigned Region Server of one of the handlers fails, log replaying hangs because the handler keeps waiting on the response of the

Table 2: The comparison of HangFix and manual fixing.

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

dead server. HangFix identifies the bug as the Pattern 4 bug because log replaying is a blocking call. HangFix's patch terminates the log replaying call directly by throwing the exception. Compared with HangFix's patch, the manual patch enables the handler to re-route the regions to another live Region Server. Therefore, log replaying job moves forward on the newly assigned Region Server.

4.2.2 Synchronization-Related Bug Patterns. Besides HangFix's four patterns, we found four other bug patterns that are related to synchronization and concurrency, i.e., missing unlock, race-induced infinite loop or blocking, and deadlock.

Missing unlock: These bugs are caused by programming mistakes on synchronization operations. When one thread is holding a lock but does not release the lock upon unexpected failures, other threads are hanging on acquiring on the lock. When missing unlock bugs happen, we observe multiple threads blocked and they all wait for the lock with the same lock ID. The safest way to fix this kind of bug is to release the lock, while terminating the blocked threads cannot fix the bug completely.

Race-induced infinite loop or blocking: The root causes of these bugs are race conditions, which further causes infinite loop or blocking. HangFix cannot ensure to fix such bug completely as currently we do not target hang bugs with race conditions as their root causes. For example, in HBase-16211 bug, if clearing JMX cache and injecting the data sink are done at the same time, the two operations access the cache simultaneously, causing the race condition. The consequence is that the injected sink lost, further leading to an reading data operation hanging. HangFix's patch jumps out of blocking reading operation, without fixing the data loss or the race condition nevertheless. The correct fix for the bug is to add a lock for cache writing operations.

Deadlock: Deadlock bugs are easy to observe through stack traces. Two threads are in BLOCKED states and they are holding different locks while they are waiting for the lock held by the other thread. HangFix's patch can terminate the deadlock but it cannot fix the deadlock.

4.3 Experimental Results

To the best of our abilities, we reproduce 42 bugs falling into HangFix's four patterns, as shown in Table 1. We list the buggy system version and the detailed description. Through the empirical study, 40 out of the 42 bugs can be completely fixed. We further validate them in our experiments. Additionally, we reproduce two partially fixed bugs and present the experimental results to illustrate why HangFix cannot fix them completely.

As shown in Table 2, HangFix successfully fixes 40 out of 42 hang bugs completely in our benchmarks, including 15

bugs in Pattern #1, 13 bugs in Pattern #2, six bugs in Pattern #3, and eight bugs in Pattern #4.

Our experiments show that for the 40 fixed bugs, the hang bug localization tool and existing bug detection tools do not raise alarms and the patched program executes successfully without hanging or crashing using the regression test suites. For the two partially bugs, HangFix cannot restore the system state or corrupted data. In contrast, 14 out of the 42 bugs are fixed by developers with manual patches, while the remaining bugs are still open or pending for developers' check to merge patches.

To further evaluate the patches generated by HangFix, we compare them with the manual patches for the 14 fixed bugs by both approaches. We find that HangFix's patches are similar as the manual ones except for the Compress-451 bug. Its manual patch throws runtime exceptions after identifying the `buffersize` variable is misconfigured to be non-positive, while HangFix throws an `IOException` which can be properly handled by the Compress system without interrupting the program's execution. We applied the manual patch on the buggy Compress program, ran our bug-triggering test case, and found that the program with the manual patch crashed after we triggered the bug.

HangFix fixed the 40 bugs completely in seconds. HangFix supports inter-procedural analysis using either iterative `BodyTransformer` or `SceneTransformer` from Soot, which decides the fixing time of the bugs. `BodyTransformer` is faster but can fail sometimes while `SceneTransformer` is relatively slower but always succeeds. HangFix first generates the patch using iterative `BodyTransformer` and evaluates the patch. If the patched code cannot fix the bug, HangFix then uses the `SceneTransformer` approach. For example, HangFix generates the patch for the Hadoop-15417 bug using `SceneTransformer` approach, which results in a longer fixing time (22 seconds) than other bugs. Note that it usually takes several weeks or even longer for developers to manually fix the bugs. It might be unfair to directly compare HangFix's patch generation time with the bug's manual resolve time because HangFix is an automatic fixing tool while manual patches involve human efforts. However, we believe that HangFix can help developers to efficiently fix hang bugs in massive cloud systems.

After adopting HangFix's patch, we measure the additional performance overhead by running the same workload again. We observe that the overhead of HangFix's patch is within 1%. Compared with the original root cause function, HangFix's patch only adds a checker (e.g., return value checker or configuration parameter checker) to the function, which imposes little overhead.

4.3.1 Two Partially Fixed Bugs. We discuss the two bugs partially fixed by HangFix, i.e., HDFS-4882 and HDFS-5438,

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

393 private synchronized void checkLeases() {
      ...
395  for(; sortedLeases.size() > 0; ) {
396   final Lease oldest = sortedLeases.first();
      ...   //p is a file's lease path
   +  int index = oldest.getPaths().size();
412   if(fsnamesystem.internalReleaseLease(p, ...)) {
413    LOG.info("...");
414    removing.add(p);//remove p from sortedLeases
   +   index -= 1;
416   } else {
417    LOG.info("...block recovery for file " + p);
418   }
   +  if(index == 0) removing.add(p); //restore stride
      ...
429 }}
```

Figure 10: Example of a pattern #3 hang bug which cannot be fixed by HangFix. A corrupted file `f` associated with the lease path `p` makes the `internalReleaseLease` function fail for recovering the lease for `f`. When it happens, `p` is not removed from `sortedLeases` (skip updating loop index), LeaseManager keeps recovering lease for the file f endlessly. "⟶" represents the function call invocation, while "- ->" represents the control flow. "+" means added code, representing the patch generated by HangFix.

in detail. HangFix cannot fix them completely because HangFix cannot restore corrupted data, which can further cause other hang problems.

HDFS-4882 (Pattern 3): As shown by Figure 10, HangFix inserts the loop index updating operations for stride restoration in the patch after line #418. HangFix's patch enables HDFS system to jump out of the infinite loop inside the hanging function `checkLeases()`. However, the outer loop between line #370 and line #388 is processing the data. Since HangFix cannot restore the corrupted data, HDFS falls into another infinite loop to process the corrupted file block endlessly.

HDFS-5438 (Pattern 4): HangFix successfully inserts the timeout settings to break an infinite polling loop. However, this patching strategy can only move forward the hanging function `completeFile()`. It cannot restore the corrupted blocks in the pipeline recovery. As a result, corrupted replicas are accepted causing another missing unlock problem in the `DFSInputStream.fetchBlockByteRange()` function.

5 RELATED WORK

In this section, we compare HangFix with most related work.

Automatic bug fixing: Previous work has proposed automatic bug fixing solutions for different bugs. AFix [23]

fixes atomicity violation bugs by quarantining critical sections using locks. CFix [24] fixes concurrency bugs by enforcing the order relationship of synchronization operations to prevent the buggy interleaving. ClearView [33] fixes invariant violation bugs by enforcing the buggy invariant to be true after changing its control state and control flow.

TFix [22] proposes a drill-down bug analysis approach to identify timeout bug's root cause and suggest correct timeout values. DFix [28] adopts the rollback and fast-forward strategy to fix distributed timing bugs.

Tufano et al. [36] applied an encoder/decoder model to mine the existing patches and automatically generate new patches. Genprog [27] is a search-based genetic programming approach for automated program repairs. SemFix [30] is a semantic-based program repair tool, which derives repair constraints from a set of tests and solves the repair constraints to generate a valid repair. Assure [35] fixes runtime faults by restoring program execution to a rescue point where error-handling is performed to recover the program execution.

Ares [18] recovers the program from runtime unexpected errors with the program's existing error-handlers. Ares synthesizes a number of error-handlers and selects the most promising one via virtual testing techniques. Gulwani et al. [19] proposed an automated program repair algorithm to use the existing correct student solutions to provide feedback for incorrect ones in programming education. Remix [16] leverages Intel's performance counter techniques to detect and repair false sharing bugs in multithreaded programs. Huron [26] presents a hybrid false sharing and repair framework with a low overhead.

Compared to those existing bug auto-fixing schemes, HangFix focuses on developing a root cause pattern driven approach to auto-patching software hang bugs in production cloud systems.

Hang bug detection: Previous work has been done to detect software hang bugs. Hang doctor [10] detects soft hangs at runtime to address the limitations of offline detection. PerfChecker [29] and HangWiz [38] automatically detect soft hang bugs by searching the application code for known blocking APIs. Cotroneo et al. [12] proposed to detect hangs in software by monitoring the response time of user actions. TScope [21] detects hang problems caused by missing timeout settings or misued timeout mechanisms. DScope [14] focuses on detecting data corruption induced software hang problems.

Tools also exist to detect hang bugs caused by inefficient loops. Jolt [11] dynamically detects infinite loops by checking each loop iteration's run-time state. Carburizer [25] statically analyzes device driver code and identifies infinite driver-polling problems.

BLeak [37] automatically debugs the memory leak problem which increases garbage collection frequency and overhead, further degrading responsiveness. Faddegon et al. [17] presented a computation tree generaration method that requires only a simple tracing library, for debugging any Haskell programs. CLARITY [32] applies static analysis to identify redundant traversal bugs, which cause serious performance problems, e.g., system hanging. DeadWait [34] constructs program representation to capture control flow and identify deadlocks in asynchronous C# programs.

In comparison to previous hang bug detection schemes, HangFix focuses on auto-fixing a detected hang bug that is triggered in production cloud environments. HangFix can leverage those hang bug detection schemes for both patch generation triggering and patch validation.

6 CONCLUSION

In this paper, we have presented HangFix, a new hang bug fixing framework for automatically patching a hang bug that is detected in production cloud environments. HangFix leverages both dynamic and static analysis techniques to localize hang functions and identify likely root cause patterns. HangFix then generates corresponding patches to fix the hang bug automatically. We have implemented a prototype of HangFix and evaluated it on 42 real-world software hang bugs in 10 commonly used cloud server systems. HangFix successfully fixes 40 out of 42 hang bugs within seconds. We have also conducted an empirical study over 237 real world hang bugs and found that our likely root cause patterns cover 76% of the 237 bugs and the rest 24% bugs are either synchronization or concurrency bugs. HangFix does not require application source or any application-specific knowledge, which makes it practical for production systems.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful feedback and valuable comments. This work was sponsored in part by NSF CNS1513942 grant and NSF CNS11 49445 grant. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or U.S. Government.

REFERENCES

[1] [n.d.]. Oracle jstack. https://docs.oracle.com/en/java/javase/13/docs/specs/man/jstack.html.
[2] 2014. Apache HBase. http://hbase.apache.org.
[3] 2015. AWS outage knocks Amazon, Netflix, Tinder and IMDb in MEGA data collapse. https://www.theregister.co.uk/2015/09/20/aws_database_outage/.
[4] 2015. Irreversible Failures: Lessons from the DynamoDB Outage. http://blog.scalyr.com/2015/09/irreversible-failures-lessons-from-the-dynamodb-outage/.

[5] 2017. What Lessons can be Learned from BA's Systems Outage? http://www.extraordinarymanagedservices.com/news/what-lessons-can-be-learned-from-bas-systems-outage/.
[6] 2019. Apache Cassandra. http://cassandra.apache.org/.
[7] 2019. Apache Hadoop. http://hadoop.apache.org/.
[8] 2019. Apache JIRA. https://issues.apache.org/jira.
[9] 2019. Soot: A Framework for Analyzing and Transforming Java and Android Applications. https://sable.github.io/soot/.
[10] Marco Brocanelli and Xiaorui Wang. 2018. Hang Doctor: Runtime Detection and Diagnosis of Soft Hangs for Smartphone Apps. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*.
[11] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. 2011. Detecting and Escaping Infinite Loops with Jolt. In *Proceedings of the European Conference on Programming Languages (ECOOP)*.
[12] Domenico Cotroneo, Roberto Natella, and Stefano Russo. 2009. Assessment and Improvement of Hang Detection in the Linux Operating System. In *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems (SRDS)*.
[13] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. 2018. Understanding Real World Timeout Problems in Cloud Server Systems. In *Proceedings of IEEE International Conference on Cloud Engineering (IC2E)*.
[14] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. 2018. DScope: Detecting Real-World Data Corruption Hang Bugs in Cloud Server Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.
[15] Daniel J. Dean, Peipei Wang, Xiaohui Gu, Willam Enck, and Guoliang Jin. 2015. Automatic Server Hang Bug Diagnosis: Feasible Reality or Pipe Dream?. In *Proceedings of IEEE International Conference on Autonomic Computing (ICAC)*.
[16] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: online detection and repair of cache contention for the JVM. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 251–265.
[17] Maarten Faddegon and Olaf Chitil. 2016. Lightweight computation tree tracing for lazy functional languages. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 114–128.
[18] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. 2016. Automatic Runtime Recovery via Error Handler Synthesis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
[19] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 465–480.
[20] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What Bugs Live in the Cloud?: A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.
[21] Jingzhu He, Ting Dai, and Xiaohui Gu. 2018. TScope: Automatic Timeout Bug Identification for Server Systems. In *Proceedings of IEEE International Conference on Autonomic Computing (ICAC)*.
[22] Jingzhu He, Ting Dai, and Xiaohui Gu. 2019. TFix: Automatic Timeout Bug Fixing in Production Server Systems. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*.
[23] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[24] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[25] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. 2009. Tolerating Hardware Device Failures in Software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*.
[26] Tanvir Ahmed Khan, Yifan Zhao, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2019. Huron: hybrid false sharing detection and repair. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 453–468.
[27] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72.
[28] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi Gunawi, and Shan Lu. 2019. DFix: Automatically Fixing Timing Bugs in Distributed Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[29] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*.
[30] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*.
[31] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *ICSE*.
[32] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 369–378.
[33] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*.
[34] Anirudh Santhiar and Aditya Kanade. 2017. Static deadlock detection for asynchronous C# programs. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 292–305.
[35] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. 2009. ASSURE: Automatic Software Self-healing Using Rescue Points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[36] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*.
[37] John Vilk and Emery D Berger. 2018. Bleak: automatically debugging memory leaks in web applications. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 15–29.
[38] Xi Wang, Zhenyu Guo, Xuezheng Liu, Zhilei Xu, Haoxiang Lin, Xiaoge Wang, and Zheng Zhang. 2008. Hang Analysis: Fighting Responsiveness Bugs. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*.