CloudRaid: Hunting Concurrency Bugs in the Cloud via Log-Mining

Jie Lu
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
China
lujie@ict.ac.cn

Feng Li
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Institute of Information Engineering, Chinese Academy of Sciences
China
lifeng2005@ict.ac.cn

Lian Li*
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
China
lianli@ict.ac.cn

Xiaobing Feng
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
China
fxb@ict.ac.cn

ABSTRACT

Cloud systems suffer from distributed concurrency bugs, which are notoriously difficult to detect and often lead to data loss and service outage. This paper presents CloudRaid, a new effective tool to battle distributed concurrency bugs. CloudRaid automatically detects concurrency bugs in cloud systems, by analyzing and testing those message orderings that are likely to expose errors. We observe that large-scale online cloud applications process millions of user requests per second, exercising many permutations of message orderings extensively. Those already sufficiently-tested message orderings are unlikely to expose errors. Hence, CloudRaid mines logs from previous executions to uncover those message orderings which are feasible, but not sufficiently tested. Specifically, CloudRaid tries to flip the order of a pair of messages $< S, P >$ if they may happen in parallel, but S always arrives before P from existing logs, i.e., excercising the order $P \nrightarrow S$. The log-based approach makes it suitable to live systems.

We have applied CloudRaid to automatically test four representative distributed systems: Apache Hadoop2/Yarn, HBase, HDFS and Cassandra. CloudRaid can automatically test 40 different versions of the 4 systems (10 versions per system) in 35 hours, and can successfully trigger 28 concurrency bugs, including 8 new bugs that have never been found before. The 8 new bugs have all been confirmed by their original developers, and 3 of them are considered as critical bugs that have already been fixed.

*Corresponding author

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236071

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Cloud computing**;

KEYWORDS

Distributed Systems, Concurrency Bugs, Bug Detection, Cloud Computing

ACM Reference Format:
Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. CloudRaid: Hunting Concurrency Bugs in the Cloud via Log-Mining. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236071

1 INTRODUCTION

Cloud computing is now mainstream. Modern online applications, from social networking and communication to banking, now play an important part of our daily life. Distributed systems, such as scale-out computing frameworks [7, 47], distributed key-value stores [15, 28], scalable file systems [15, 28], and cluster management services [47], are the fundamental building blocks of cloud applications. However, distributed systems are notoriously difficult to get right. It is too complicated for the programmers to correctly reason and handle concurrent executions on multiple machines. There are widely existing concurrency bugs in real-world distributed systems, which often lead to data loss and sometimes service outage [20, 50]. For example, on November 18th 2014, the Microsoft Azure cloud hosting service went down for around 24 hours, due to a bug in a server-side software update.

Distributed concurrency bugs are triggered by untimely interaction among nodes, i.e., unexpected message orderings [18, 30]. This fact has motivated a large body of research on distributed system model checkers [19, 29, 33, 41], which detect hard-to-find bugs by systematically exercising all possible messages orderings. Theoretically, these model checkers can guarantee reliability when running

a same workload. However, distributed system model checkers face the state-space explosion problem [29]. Despite recent advances [29], it is still difficult to scale these tools to many large real-world applications. For example, in our experiments of running the WordCount workload on Apache Hadoop2/Yarn, 5,495 messages are involved and it is impractical to exhaustively test all possible message orderings in a timely manner.

This paper proposes a novel strategy in battling distributed concurrency bugs. We do not try to exhaustively exercise all possible message orderings. Instead, we address a different question: which message orderings are likely to trigger an error? Hence, our approach analyzes suspicious message orderings and only test those orderings that are likely to expose errors. The approach is not sound and does not guarantee free of concurrency bugs. However, it is very effective in detecting distributed concurrency bugs, as highlighted in our experiments. It is also simple and can be easily adopted by live systems.

Which message ordering is likely to trigger an error? This question is key to our approach. We address the question based on the following observations:

- $Observation_1$. The errors triggered by different message orderings often share a common root cause pattern: their corresponding message handlers access some shared objects inconsistently, when given different orders.
- $Observation_2$. Large-scale online applications process millions of user requests per second. Many permutations of message orderings have already been extensively tested and exercised in these live systems. Those sufficiently-tested message orderings are unlikely to expose errors.

Hence,we can learn from previous executions to uncover these message orderings that are likely to expose errors. Since we harness the rich execution history from live systems, a non-intrusive log-based approach is desirable. Modern cloud applications often provide a rich set of runtime logs, which record important messages and events to help with the diagnosis and monitoring of online systems. Our approach mines logs from previous executions, to uncover those message orderings which are feasible but not yet sufficiently tested. The log-based approach makes it suitable to live systems, where intrusive instrumentation is often not an option.

We develop CloudRaid, a new tool to effectively detect distributed concurrency bugs. CloudRaid automatically extracts sequences of important communication events from existing run-time logs. Permutations of these event orderings will be further tested if they are feasible but not yet exercised. A dynamic trigger is employed to exercise and test the selected message orderings at runtime. Previous studies [30] show that more than 60% distributed concurrency bugs can be triggered by a single untimely message delivery. Hence, we focus on the order between a pair of messages only. CloudRaid tries to flip the order of a pair of messages $< S, P >$ if they may happen in parallel, but S always arrives before P from existing logs, i.e., exercising the order $P \nrightarrow S$.

We have applied CloudRaid to test four representative distributed systems: Apache Hadoop2/Yarn [47], HDFS [5], HBase [15], and Cassandra [28]. CloudRaid can be easily adopted. The system under testing can run as is lively, without modification. In a separate testing phase, our dynamic trigger performs minimal instrumentation to test a specific message ordering. In our evaluation, we randomly choose 40 different versions of these systems (10 versions per each system), and ran 4 different workloads in total on these systems. CloudRaid ran 3200 times all together in 35 hours, where each run tries to exercise a specific message ordering. The 3200 runs successfully triggered 28 bugs (with no false positives), including 8 new bugs that have never been found before. The 8 new bugs have all been confirmed by the original developers, and 3 of them are considered as critical bugs and have already been fixed.

Contributions. This paper makes the following contributions:

- We propose a new approach to effectively detect concurrency bugs in distributed systems. Our approach avoids unnecessary repetitive tests by harnessing the rich log information in previous running histories, which can drastically improve efficiency.
- We develop CloudRaid, a simple yet effective tool to test distributed systems. CloudRaid targets live systems by automatically analyzing run-time logs, without instrumentation. It can be easily adopted and is very effective in detecting distributed concurrency bugs.
- We extensively evaluated CloudRaid using four representative distributed systems: Apache Hadoop2/Yarn, HBase, HDFS, and Cassandra. CloudRaid can finish testing 40 different versions of the 4 systems (with 4 workloads in total) in 35 hours, and can successfully detect 28 concurrency bugs. Among them, there are 8 new bugs, including 3 critical bugs which have already been fixed by their original developers.

The rest of the paper is organized as follows. Section 2 illustrates our approach using a real-world example. We present the design and implementation of CloudRaid in Section 3 and evaluate its efficiency and effectiveness in Section 4. Section 5 reviews related work and Section 6 concludes the paper.

2 AN ILLUSTRATION EXAMPLE

The example in Figure 1 depicts the common procedure to create a new task in Hadoop MapReduce. There is a concurrency bug MAPREDUCE-3656 in this basic procedure.

The Bug. The two messages, (5) and (9), that trigger the bug are highlighted in red. In normal execution (Figure 1), the remote procedure call (RPC) to `startContainer` finishes execution quickly. Hence, the `ASSIGNED` event (message (5)) is always dispatched and handled before the `UPDATE` event (message (9)). However, there is no happen-before order between the two messages. If the `UPDATE` event arrives before the `ASSIGNED` events (due to unexpected delays in T_1, e.g., insufficient resources in the underlying machine running AM), an error is triggered and the task cannot be created.

The Root Cause. The event handler implements a state machine for each task, to update its status according to the incoming events. The state machine expects to always process the `UPDATE` event after the `ASSIGNED` event. Otherwise, an error will be thrown which leads to job fail. The fix is to introduce smart synchronizations to guarantee that the `ASSIGNED` event always arrives before the `UPDATE` event.

Among all these messages, the message pair <(5), (9)> is the root cause of the error. Both messages are handled by a same event

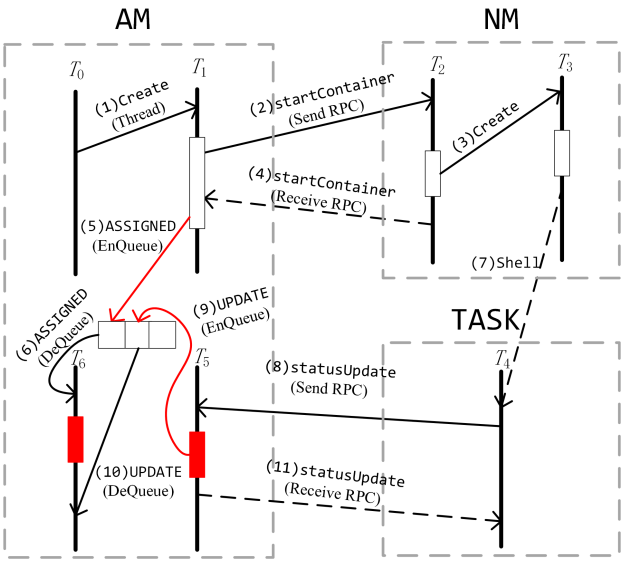

Figure 1: A real-world example to start a new task in Hadoop2. AM is the application manager node, NM is the node manager node, and TASK is the node to run the task. (1) Thread T_0 in AM create a new thread T_1, (2) T_1 invokes the remote procedure `StartContainer` to start a container on NM (Thread T_2), (3) In the `StartContainer` method, T_2 creates another thread T_3, (4)the RPC (remote procedure call) to `StartContainer` returns to T_1, (5) After returning from `StartContainer`, T_1 sends a task `ASSIGNED` event to the event queue, (6) the `ASSIGNED` event is dispatched to the event handler T_6, the TASK state is updated to `ASSIGNED`, (7) T_3 starts a new process to run the task on node TASK using shell script (Thread T_4), (8) T_4 invokes the remote procedure `statusUpdate` on AM (Thread T_5), (9) In T_5, the `statusUpdate` method sends a task `UPDATE` event to the event queue, (10) the `UPDATE` event is dispatched to the event handler T_6, to update the TASK state to `UPDATED`, (11) Method `statusUpdate` returns to T_4.

handling method, to update the same variable for task status. How can we automatically pick the message order (9) ↣ (5), among all the rest messages? Let us dig into the technical details.

2.1 Source Code and Runtime Logs

Figure 2 gives the abstracted code snippet of our illustration example. Those lines sending messages are highlighted in red. We call those source locations sending and handling messages *static messages*. Hereafter, we use the notation M for a static message, and M_i for its dynamic instance.

There are 3 common patterns of static messages, thread creation (message (1) in line 3), RPC (remote procedure call, message (2) in line 11), and event dispatch (message (5) and message (9) in line 12 and line 38, respectively). For simplicity, the code snippets for sending messages (3) and (7) are not given. The event handling method `EventHandler.handle` (line 19) calls method `StateMachine.doTransition`, which invokes different callback functions to handle different types of events. Here we present a simplified version with callbacks inlined.

The boxed lines log static messages. All messages, except for the RPC return (message (4) and (11)), and the call to shell script (message(7)), are logged. A message is often logged at the entry of its corresponding handler. Message (9) (line 38) follows immediately after message (8) (RPC to method `statusUpdate` in line 36). Hence, a common log (line 37) is introduced to serve both messages for better performance. In this case, we group the two static messages together, denoted as (8,9).

The logs consist of constant strings and values of variables. The above code snippet will execute multiple times at run time, resulting in multiple dynamic instances per static message, as well as multiple log instances. Values of variables in the log instances are used to distinguish each dynamic instance. Figure 3 shows the simplified runtime logs, where the code snippet is executed twice.

2.2 Methodology

Ideally, we would like to precisely recover runtime message sequences from existing logs, as annotated in Figure 3. Each log instance is mapped to one static message (or a grouped static message). Log instances from the same run are grouped together in order. In reality, we perform source code analysis and log analysis together, to recover such message sequences. We statically analyze how static messages are handled and logged. Runtime log instances can then be mapped to static messages with static analysis information. We group logs from the same run together by analyzing the relation between logged variable values, based on static dependence analysis and their runtime values. Section 3 gives the technical details.

The recovered message sequences are then mutated for further testing. In this paper, we focus on the order between a pair of static messages $< P, S >$, where P and S may happen in parallel. Some message pairs follow a strict happen-before order, e.g., <(1),(2)>, <(2),(3)>, and <(2),(5)>. The order between them can not be mutated. Our observations in Section 1 provide the basic guide lines to select a message ordering $P \nrightarrow S$, as follows:

- $Rule_1$. Runtime log instance P_i and S_i must log *related* runtime values of ID variables.
- $Rule_2$. The order $P_i \nrightarrow S_i$ have not been exercised, where P_i and S_i are runtime instances with matching values of ID variables.

$Observation_1$ leads to $Rule_1$. Distributed systems frequently use values of ID variables as indexes to access shared resources. Thus, messages logging completely unrelated values of ID variables are unlikely to access a common shared objects, and unlikely to expose errors. $Rule_2$ discards these message orderings that have already been exercised, according to $Observation_2$.

In our example, the message pairs <(3), (5)> and <(8,9),5> may happen in parallel. All messages record values of related variables in their logs ($Rule_1$). The message orderings (3) ↣ (5), 5 ↣ (3), and (5) ↣ (8,9) have already been tested, according to the log information. Hence, we will select the order (8,9) ↣ (5) for further testing. The error can then be triggered.

Discussion. Our method mutates the order between a pair of messages only. We do not target bugs that occur due to multiple messages being out of order. Previous studies [30] have shown that most distributed concurrency bugs are triggered by a single untimely message, and only 26% of bugs require more than 2 messages.

```
// Thread T0 in AM
1  public void ContainerLauncherImpl.serviceStart() {
2      Runnable t = createEventProcessor(new ContainerLaucherEvent());
3      this.launcherPool.execute(t);  //  Message (1)
4  }
5  public void ContainerLauncherImpl.createEventProcessor(ContainerLauncherEvent event) {
6      return new EventProcessor(event);
7  }

// Thread T1 in AM, handler of Message(1)
8  public void EventProcessor.run() {
9      LOG.info("Launching " + this.taskAttemptID);
10     ContainerManagementProtocolPBClientImpl proxy = getCMProxy(this.containerMgrAddress);
11     StartContainerResponse response = proxy.startContainer(new StartConReq(...)); // Message (2)
12     this.dispatcher.handle(new TaskAttemptContainerLaunchedEvent(this.taskAttemptID...));   // Message (5)
13 }

// Thread T2 in NM, handler of Message(2)
14 public StartConRes ContainerManagerImpl.startContainer(StartConReq req) {
15     ID containerID = req.getConLauContext().getConId();
16     LOG.info("Start request for " + containerID);
17     . . .  // create thread T3 via thread pool, message(3)
18 }

// Thread T6 in AM, handler of Message(5) and Message(9) (dispatched by the event dispatcher)
19 public void EventHandler.handle(TaskEvent event){
20     TaskTAttemptEvent ev = (TaskTAttemptEvent) event;
21     if (this.oldState== ASSIGNED&& ev.getType().equals(TA_CONTAINER_LAUNCHED)) {
22         // Handle message (5)
23         TaskAttempt attempt = ev.getTaskAttempt();
24         LOG.info("TaskAttempt: ["+attempt.attemptId+"] using containerId: ["+attempt.containerID);
25         . . .  // Update task status
26     } else if (this.oldState== ASSIGNED && ev.getType().equals(TA_CONTAINER_UPDATE)) {
27         // Handle message (9)
28         . . .
29     }   // Other cases
30 }

// Thread T3 in NM, handler of Message(3)
31 public void LocalizerRunner.run() {
32     nmPrivateCTokensPath = getLocalPathForWrite(this.localizerId);
33     LOG.info("Writing credentials to the nmPrivate file " + nmPrivateCTokensPath.toString());
34     . . . // create TASK via Shell script, message (7)
35 }

// Thread T5 in AM, handler of Message(8)
36 public void TaskAttemptListenerImpl.statusUpdate(TaskAttemptID taskAttemptID) {
37     LOG.info("Status update from " + taskAttemptID);
38      this.dispatcher.handle(new TaskAttemptStatusUpdateEvent(. . .)); // Message (9)
39 }
```

Figure 2: Abstracted code snippet of the example in Figure 1.

This study motivates our approach. Our method can effectively detect distributed concurrency bugs in real-world cloud systems, as demonstrated in our evaluation.

3 THE CLOUDRAID APPROACH

We implement CloudRaid in WALA [2] via a series of sub-analyses (Figure 4). *Communication analysis* statically analyzes how static messages are handled and logged. Similar to [48, 52], we represent logging patterns of static messages as regular expressions. Then,

```
1 Launching attempt_1514878932605_0001_m_000009_0    // Message (1)
2 Start request for container_1514878932605_0001_01_000011 // Message (2)
3 Writing credentials to the nmPrivate file
$HADOOP_HOME/nm-local-dir/nmPrivate/container_1514878932605_0001_01_000011.tokens // Message (3)
4 TaskAttempt: [attempt_1514878932605_0001_m_000009_0] using containerId: [container_1514878932605_0001_01_000011 //Message (5)
5 Status update from attempt_1514878932605_0001_m_000009_0 // Message (8), immediately followed by (9)

6 Launching attempt_1514878932605_0002_m_000007_0 // Message (1)
7 Start request for container_1514878932605_0002_01_000009 // Message (2)
8 TaskAttempt: [attempt_1514878932605_0002_m_000007_0] using containerId: [container_1514878932605_0002_01_000009] //Message (5)
9 Writing credentials to the nmPrivate file
$HADOOP_HOME/nm-local-dir/nmPrivate/container_1514878932605_0002_01_000009.tokens // Message (3)
10 Status update from attempt_1514878932605_0002_m_000007_0 // Message (8), immediately followed by (9)
```

Figure 3: Simplified runtime logs of the example in Figure 1.

Table 1: Static Messages $< C, F, L >$.

Messages	Client site C	Message handler F	Logging pattern L
(1)	3	`EventProcessor.run`	Launching attempt_*
(2)	11	`ContainerManagerImpl.startContainer`	Start request for container_*
(3)	-	`LocalizerRunner.run`	Writing . . . $HADOOP.../container_*.tokens
(5)	12	`EventHandler.handle`	TaskAttempt: [attempt_*] . . . [container_*]
(8)	-	`TaskAttemptListenerImpl.statusUpdate`	Status update from attempt_*
(9)	38	`EventHandler.handle`	Status update from attempt_*

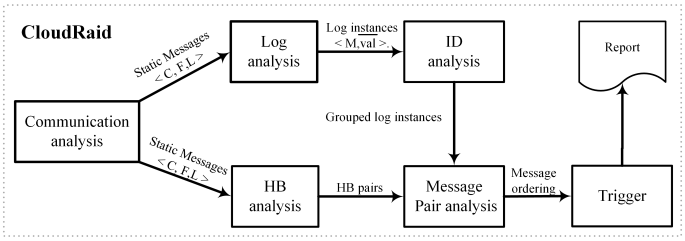

Figure 4: Architecture of CloudRaid.

Log analysis uses the logging patterns to map each runtime log instance to a static message. *ID analysis* analyzes relations between logged values, according to dependences between logged variables and their run time values (from Log analysis). Messages from the same run can then be distinguished from other runs and grouped together. *HB analysis* statically analyzes the happen-before order between static messages. Analysis results from HB analysis and ID analysis are used in *Message Pair analysis*, to select message orderings for further testing. Finally, *Trigger* will instrument the source code to exercise the selected message orderings.

3.1 Communication Analysis

Communication analysis is the basis for all subsequent analyses. It represents each static message as a tuple of 3 elements $< C, F, L >$, where C is the client site to send the message, F is the corresponding message handler, and L is the regular expression expressing its logging pattern. Table 1 gives the static messages for our illustration example (client site C is represented using source line number). We target 3 common patterns of static messages: thread creation, RPC, and event dispatch.

3.1.1 Thread Creation. The client site C is the call site to `t.start()` or `ThreadPoolExecutor.execute(t)`, where `t` is a Runnable or Thread object. The message handler F is the `run()` method of the thread object referenced by `t`. To locate the message handler F, we backward slice the program to find the object that `t` points to. Instead of using a standard pointer analysis [31, 32, 42, 45], we simply follow the def-use chains since the thread object is often created right before its execution.

In Figure 2, the call site `this.launcherPool.execute(t)` (line 3) is the client to start a thread. Slicing `t` backward, we can reach the object created at line 6 (`new EventProcessor`). Hence, the message handler F is `EventProcessor.run` (line 8). Static message (1) is thus represented as <3, `EventProcessor.run`, L>, where 3 is the line number and L is the logging pattern to be analyzed.

3.1.2 RPC. RPC allows users to call a remote procedure in the same way as calling a local function. The client site C is the local call site, and the message handler F is the invoked remote procedure. In common practices (e.g., google protobuf [1]), RPC is realized via a client class and a corresponding server class with the same interface. To identify RPC, we require the user to specify the RPC client classes, as well as the RPC server classes. A RPC server class can be matched to its corresponding RPC client class by checking their public APIs. We automatically recognize classes that wrap RPC client classes using the delegation design pattern [13]. Thus, given a RPC client site, we can easily find its RPC server class and the corresponding remote procedure to handle this message.

In Figure 2, `ContainerManagementProtocolPBClientImpl` is a RPC client class. The call to `proxy.StartContainer` (line 11)

is the client site of RPC. Its corresponding RPC server is identified as `ContainerManagerImpl`. Hence, the method handler F is `ContainerManagerImpl.startContainer` (line 14). Static message (2) is <11, `ContainerManagerImpl.startContainer`, L>.

3.1.3 Event Dispatch. The client site C is the call site to enqueue an event, and the message handler F is the method to handle the dispatched event. We abstract away the complicated implementation details of the asynchronous event dispatch mechanism. Here we require the user to specify the methods to enqueue an event, and the methods to handle an event. Distinct types of events may be handled by different handlers. In this case, we check the type of the enqueued event, and the type of the dispatched event (formal arguments of the event handler), to match the handler with an enqueued event of the same type.

In Figure 2, `Dispatcher.handle` is the method to enqueue an event, and `EventHandler.handle` (line 19) is the method to handle a dispatched event. There are two events, message (5) and message (9). Their client sites are the call sites to `Dispatcher.handle` method at line 12 and 38, respectively. Both events are handled by the same handler. Hence, the two messages share a common message handler `EventHandler.handle` (line 19).

3.1.4 Message Logging Pattern. We firstly locate a log point (e.g., call to `Log.info` and their wrappers) for each static message. In the simple case, a message is logged in the entry block of its message handler (when the message is received and handled). For example, the log point for message (1) (<3, `EventProcessor.run`, L>) is line 9, and the log point for message (2) (<11, `ContainerManagerImpl.start-Container`, L>) is line 16. Hence, we search the entry block of the message handler F, as well as the entry blocks of these methods invoked in the entry block of F, for a log point. If there exists multiple log points for a static message, we group them together as one log point.

Some systems implement a complicated handler for different types of events. In Hadoop2, the event handler implements a state machine which executes different cases and invokes different callback functions, according to the incoming event type and the current state (method `EventHandler.handle` in Figure 2, line 19). In this case, we require the user to specify the transition rules of the state machine, e.g., which callback function handles which type of event. We can then statically check the type of the enqueued event (at the client site) to find a matching case in the handler, and locate the corresponding log point.

In Figure 2, message(5) (<12, `EventHandler.handle`, L>) and message (9) (<38, `EventHandler.handle`, L>) are handled by the same method `EventHandler.handle` (line 19). The method executes different cases according to the incoming event type. Hence, we check the event type at the client site for a matching case. Message (5) enqueues an event of type `TaskAttemptContainerLauched-Event`, which sets its `TYPE` field to `TA_CONTAINER_LAUNCHED` in the constructor. By analyzing this field, we can find a matching case (lines 22-25). The log point at line 25 is located. Similarly, we can deduce that message (9) is handled by lines 27 and 28. There is no log point in its handler. In this case, we will search for a log point at the client site C, in the basic block containing C. Hence, line 37 is regarded as the log point of message (9). Messages share the same log point are grouped together. In our example, message (9) and message (8) are grouped together.

The message logging patterns can then be extracted at each log point. Following previous work [48, 52], logging patterns are expressed using regular expressions. We analyze the logging statement at each log point, and the `tostring` method of logged variables, to statically extract the constant strings in the log message. Runtime values of logged variable are denoted as *. Table 1 summarizes the logging patterns for each message in Figure 2.

3.1.5 Discussion. We design our communication analysis in such a way that minimal user specification is required. In our approach, for RPC, we require the user to specify the RPC client classes and server classes. For event dispatch, we require the user to specify the event enqueue method, the event handler, and the call back functions (if there are any) for different types of events. The communication analysis then automatically analyzes each message client site, identifies its corresponding handler, and locates the right log point to extract its logging pattern. The precision of communication analysis can be further improved if the user can provide detailed annotations to specify the client site, the message handler, and the logging pattern for each message. Alternatively, we could get such precise information via instrumentation and profiling. In the four different distributed systems we studied, our communication analysis can correctly extract the logging patterns for the majority of messages, without loss of precision.

3.2 Log Analysis

Log analysis tries to match each runtime log instance to a message logging pattern. A log instance is represented as a tuple $< M, \overline{Val} >$, where M is the static message, and $\overline{Val}$ records the runtime values of logged variables. Table 2 gives the representation of each runtime log instance in Figure 3. Logs are ordered according to their time stamps, the time stamps are calibrated to a centralized time to compensate for the time differences on distinct nodes, as in [35].

We adopt the approach in [48] to match each runtime log instance to a static message logging pattern efficiently. A reverse index is built as a hash for each logging pattern, which can be used to quickly calculate a matching score for each log instance. The higher the score, the more likely it is a match. For each log instance, we select 10 message logging patterns with the highest scores, then parse each log instance according to the 10 logging patterns, to find a exact match. For each log instance, we also record the runtime values of its logged variables, as shown in Table 2.

3.3 ID Analysis

ID analysis organizes related log instances in a hierarchy structure, based on *ID Values*, i.e., runtime values of *ID variables*. The hierarchy structure represents tasks and sub-tasks. In distributed systems, values of ID variables are commonly used to distinguish distinct requests and tasks. These variables are wrapped in messages and propagated to different nodes and threads. Therefore, we regard a variable as an *ID variable* if 1) it is propagated from a message, i.e., formal arguments of message handlers or fields of runnable objects, and variables accessible from them (via direct or indirect field dereferencing); and 2) the variable is printed in logs. It is very difficult to precisely analyze the propagation of ID variables

Table 2: Log instances $< M, \overline{val} >$.

Log Instances	Static Message	Runtime Values
1	(1)	attempt_1514878932605_0001_m_000009_0
2	(2)	container_1514878932605_0001_01_000011
3	(3)	...container_1514878932605_0001_01_000011
4	(5)	attempt_1514878932605_0001_m_000009_0 ; container_1514878932605_0001_01_000011
5	(8,9)	attempt_1514878932605_0001_m_000009_0
6	(1)	attempt_1514878932605_0002_m_000007_0
7	(2)	container_1514878932605_0002_01_000009
8	(5)	attempt_1514878932605_0002_m_000007_0; container_1514878932605_0002_01_000009
9	(3)	...container_1514878932605_0002_01_000009
10	(8,9)	attempt_1514878932605_0002_m_000007_0

statically when complicated pointer and fields dereferencing are involved. Hence, we statically analyze an initial set of ID variables, and use their runtime values to group log instances with same ID values together. These variables which are propagated from formal arguments or runnable object fields to a log point, via direct assignments or field dereferences, are included in the initial set of ID variables.

In our example (line 33), we cannot statically determine whether the logged variable `nmPrivateCTokensPath` is propagated from formal arguments or not. It is not included in the initial set of ID variables. However, the log can still be grouped according to its runtime value, which can be matched to an already existing ID value (`containerID` logged at line 16).

Definition 1. *ID value V_1 and ID value V_2 are* related *if there exists a log instance $< M, \overline{Val} >$, such that both $V_1 \in \overline{Val}$ and $V_2 \in \overline{Val}$ hold. Let $\mathcal{L}$ be the set of log instances of static message M. ID value V_1 is a sub-ID of V_2 if for any log instance $< M, \overline{Val} > \in \mathcal{L}$, $V_1 \in \overline{Val} \implies V_2 \in \overline{Val}$, but not vice versa.*

Log instances with a same ID value are grouped together to perform a *task*. A tasks is indexed with one ID value, and can be further divided into sub-tasks. Two tasks are *related* if their ID values are related. One task is a sub-task of another, if its ID variable is a sub-ID of another.

For our example, the log instances in Table 2 are organized into two groups. Each group consists of two related tasks, indexed by the runtime values `attempt_*`, and `container_*`, respectively.

3.4 HB Analysis and Message Pair Analysis

HB analysis analyzes the happen-before relation between static messages. We consider two simple types of happen-before relation. Given static message $P :< C_P, F_P, L_P >$, and $S :< C_S, F_S, L_S >$. If C_S is in method F_P, then P happens before S. If C_P dominates C_S and C_P is a RPC client site, then P happens before S. We compute the transitive happen-before relation for all static messages.

Message pair analysis selects the order $P \nrightarrow S$ for further testing. If P happens before S or S happens before P, then the order is either infeasible or always holds, and it will not be selected. We check whether P and S are related or not, and whether the order $P \nrightarrow S$ have been exercised or not, by comparing their log instances in a pair-wise manner. Given log instance $P_i :< P, - >$ and log instance $S_i :< S, - >$. If P_i and S_i belongs to the same or related tasks, then P and S are related. If P_i and S_i are related and P_i is logged before S_i, then $P \nrightarrow S$ has already been tested. The order $P \nrightarrow S$ will be selected if P and S are related, and the order has not been exercised.

3.5 Trigger and Error Report

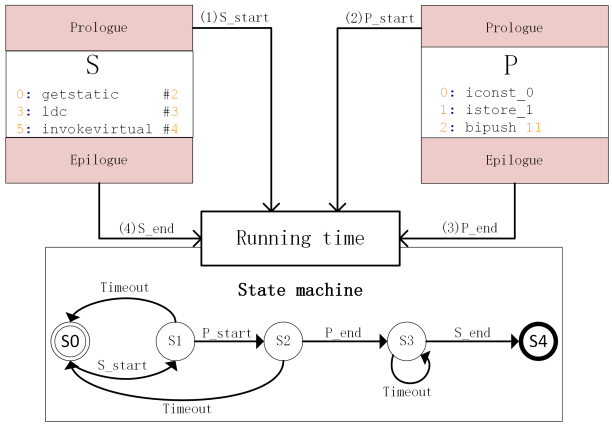

Figure 5: Trigger order $P \nrightarrow S$, i.e., flip the order $S \nrightarrow P$.

The trigger tries to exercise the selected message order $P \nrightarrow S$ by instrumenting the system in such a way that a dynamic instance S_i can wait until P_i is handled. We introduce an prologue and epilogue for both S and P. The prologue is instrumented before the message is handled, and the epilogue is introduced after the message is handled. For RPC and thread creation, the prologue and epilogue are introduced at the entry and exit of the message handler, respectively. For event dispatches, the prologue is introduced at the client site before the event is enqueued, and the epilogue is inserted at the exit of the message handler. The reason is that the prologue will block execution. If the event handler is blocked, no event can be dequeued and the event queue will soon be occupied.

As shown in Figure 5, we maintain a state machine at runtime. Initially, neither P nor S is executed. The start state is `S0`. When executing S, the prologue of S sends a $\mathtt{S}_{\mathtt{start}}$ event to our runtime. The state machine is updated to state `S1`, and S will sleep for a time interval $T + \delta$, waiting for P to be executed. The interval T is set to be the largest interval between related instances of P_i and S_i, in previous executions. We introduce a δ to compensate for the delay of our instrumented code, and the delay to handle message P. As a

result, P will have a large chance to be handled while S is waiting. When P starts to execute, the prologue of P sends a $\mathsf{P}_{\texttt{start}}$ event, and the state machine is updated to S2. After P finishes execution, the epilogue of P sends a $\mathsf{P}_{\mathsf{end}}$ event, the state is updated to S3. After waiting for $T + \delta$, the prologue of S sends a `Timeout` event , then continues execution. If P have already finished execution (state S3), the message order can be successfully exercised. The state reaches the final state S4 when S finishes execution. Otherwise, if P does not arrive in time (state S1), or has not finished execution (state S2), the runtime state is reset to its initial state, suggesting that we have not successfully trigger the order.

Finally, after execution, CloudRaid reports an error in the following 3 cases: 1) system crash, 2) job hang or fail, and 3) there exists uncommon error level logs in the log file. Currently, we do not report silent errors which lead to unexpected behaviors that are difficult to detect, e.g., silent data corruptions [10]. How to develop test oracles to automatically detect such unexpected behaviors is an important topic worth separate investigation.

4 EVALUATION

Table 3: Systems under testing.

System	# CloudRaid code changes	Workload
Hadoop2/Yarn	48	wordcount + kill
HDFS	18	putfile + reboot
HBase	25	write + node crash
Cassandra	17	write

We apply CloudRaid to four representative real-world distributed systems: Apache Hadoop2/Yarn (distributed computing framework), HDFS (distributed file system), HBase (distributed key-value stores), and Cassandra (distributed key-value stores). Table 3 presents the 4 systems. Column 2 gives the code changes required (in #LOC) to adapt CloudRaid to a new system. On average, we need to apply 27 lines of code changes for each system, to specify its communication patterns. In this experiment, we use 4 failure-triggering workload described in [30] and run the systems using their default configurations (including default logging configurations). These workloads are also common workloads, but errors may be triggered by untimely communication among nodes.

Each system runs the workload 20 times, to generate runtime logs. CloudRaid then performs its analyses using these logs. We have experimented with larger sets of logs (up to 50 runs), and no noticeable difference is observed. All the experiments are performed on a cluster with three identical nodes. Each node has a CentOS 6.5 system on an Intel(R) Xeon(R) E7-4809 processor with 32 GB of memory. The evaluation will answer the following research questions:

- **RQ1.** How accurate can CloudRaid extract message sequences from runtime logs?
- **RQ2.** How effective is CloudRaid in detecting bugs, can it detect new bugs?
- **RQ3.** How much does CloudRaid improve testing efficiency?

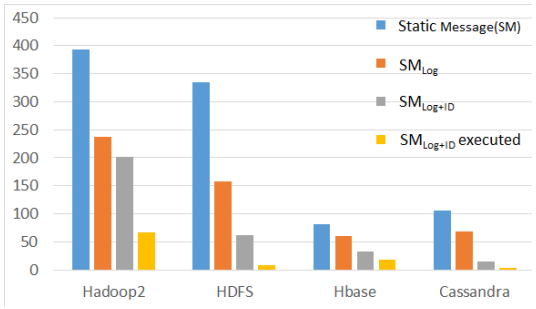

Figure 6: Number of static messages (SM), SM with logs (SM_{Log}), SM with logs and IDs (SM_{Log+ID}), and SM with logs and IDs that have been executed.

4.1 RQ1: Accuracy

Figure 6 summarizes the identified static messages in the 4 systems. There are 393 static messages in Hadoop2/Yarn, and 82 static messages in HBase. Our communication analysis can successfully analyze the logging pattern for more than 60% of static messages in all systems, except for HDFS (46.8%). We manually inspected these static messages without logging patterns. Their logs are often optimized out for performance reasons. HDFS frequently reads file systems without logging, hence a large percentage of static messages in HDFS do not have a logging pattern.

ID analysis can successfully find an ID value in 84.8% of the static messages for Hadoop2/Yarn, and in 52.5% of the static messages for HBase. ID values can effectively distinguish log instances from different runs. However, only 39.2% static messages in HDFS, and 22.1% static messages in Cassandra have an ID value. HDFS frequently invokes RPC to get or set the state of name node, without ID values. Cassandra mainly prints logs during system startup. Since we target live systems, we process logs for user requests. We further analyze these message logs without ID values: 78.4% of them print variables such as `size`, 16.5% are daemon process printing service start or stop messages, and the rest 5.1% are due to bad log quality.

Table 4: Statistics of runtime logs. "#Log instances" is number of different types of runtime log instances. "#Static messages" is the number of messages covered by runtime logs.

System	#Log instances			#Static messages	
	SM	SM_{ID}	SM + NonSM	SM	SM_{ID}
Hadoop2/Yarn	11519	8539	15813	122	67
HDFS	7777	7750	8073	22	9
HBase	9663	5753	10614	59	18
Cassandra	4417	104	14263	29	4

Table 4 gives the number of runtime log instances (Columns 2-4), and the number of static messages covered by runtime logs (Columns 5 and 6). Let us compare Column 2 with Column 4. In all benchmarks, except for Cassandra, the majority of runtime log instances record message events (72.8% for Hadoop2/Yarn, 96.3% for HDFS, 91.0% for HBase). In comparing Column2 to Column 3, we find that most message log instances also record ID values (74.1%

for Hadoop2, 99.6% for HDFS, and 60% for HBase). These systems provide valuable information for CloudRaid to accurately recover the runtime message sequences. Cassandra is an exception, with much fewer messages being logged and ID values rarely being used in the logs. The reason is that CloudRaid process request logs, while Cassandra prints most of its logs during the system startup process. As a result, only 0.73% runtime log instances in Cassandra record a message event with ID values, which is also shown in Figure 6.

The runtime log instances covers about 32% of static messages (Column 6 in Table 2, and Figure 6). The other uncovered part may need a distinct workload (e.g., alter table for HBase), or a different configuration (e.g., to execute a distinct resource scheduling model in Yarn), or it is in an error handling module difficult to reach.

Discussion. The accuracy of message sequences extracted by CloudRaid differs in different systems. Overall, Hadoop2/Yarn provides the most accurate information in its logs. CloudRaid can analyze and process 72.8% of the runtime logs, and it exercises more static messages than the other 3 benchmarks (Column 6 in Table 4). Cassandra rarely logs ID values. CloudRaid can only analyze 0.73% of runtime log instances, and cannot accurately recover its runtime message sequences from logs.

4.2 RQ2: Effectiveness

We evaluate how effective CloudRaid is in testing known bugs, as well as its ability in detecting new bugs.

Table 5: Bug detection results against TaxDC [30].

Detected	MR-3656 MR-3274 MR-4637 MR-3596 MR-2995 MR-4751 MR-4607 MR-5358 CA-5631 HBase-4539 HBase-6070 HBase-5816
Not Detected	MR-3006 MR-4099 MR-5009 MR-3721 MR-4842 HBase-6537 HBase-10257 HBase-8940

4.2.1 Finding existing bugs. We evaluate CloudRaid against the TaxDC Benchmark suite [30]. The 20 benchmarks in Table 5 are selected because we can manually trigger a failure by changing the order of a message pair in these benchmarks. We skip those benchmarks in TaxDC if we cannot reproduce the bug manually, or if it involves timely hardware failure, or if it requires to reorder multiple pairs of messages together. CloudRaid can report 12 of the 20 benchmarks automatically. In 6 of the 8 undetected benchmarks (MR-5009, MR-3721, MR-4842, HBase-6537, HBase-10257, and HBase-8940), their messages are not logged. Hence, CloudRaid fails to report errors in these benchmarks. We argue that the log quality needs to be further improved in this case, to help better diagnose this type of failure. The two benchmarks, MR-3006 and MR-4099, require instrumentation in the middle of their message handlers, which cannot triggered by CloudRaid. A more sophisticated instrumentation strategy can help to trigger such errors.

4.2.2 Detecting new bugs. We evaluate the ability of CloudRaid in detecting new bugs using the four systems in Table 3. For each system, we select 10 different versions (the latest version, the oldest version, and 8 random selected versions). We apply CloudRaid to test each version. The same bug appearing in different versions is reported as one bug.

Table 6: Bug detection results against the systems in Table 3.

System	#Bugs: new/all		
	Order Violation	Atomicity Violation	Total
Hadoop2/Yarn	6/19	1/2	7/21
HDFS	0/3	0/0	0/3
HBase	1/2	0/2	1/4
Cassandra	0/0	0/0	0/0
Total	7/24	1/4	8/28

CloudRaid can successfully find 28 bugs, including 20 already tracked bugs and 8 new bugs. Most of the bugs detected by CloudRaid are message order violation bugs (24 out of 28), which is expected. CloudRaid also detects 4 atomicity violation bugs. By reordering messages, CloudRaid can impact other message handlers and make them execute concurrently.

CloudRaid detects the most number of bugs (21 out of 28, Column 4 in Table 6) in Hadoop2/Yarn, and none in Cassandra. Cassandra is the benchmark with the least log information (Figure 2 and Table 2). The limited log information largely restricts CloudRaid's ability in detecting bugs.

Table 7: New bugs detected. All bugs are confirmed by the original developers, and 3 of them are already fixed.

Bug ID	type	status	Patched?	Symptom
YARN-6948	Order	Fixed	yes	Attempt fail
YARN-6949	Order	Unresolved	no	Wrong state
YARN-7176	Atomicity	Unresolved	yes	Cluster down
YARN-7563	Order	Unresolved	yes	Resource leak
YARN-7663	Order	Fixed	yes	Job fail
YARN-7726	Order	Unresolved	yes	Wrong state
YARN-7786	Order	Fixed	yes	Null Pointer
HBase-19004	Order	Unresolved	no	Data loss

Table 7 summarizes the 8 new bugs detected by CloudRaid. These new bugs may lead to serious failure such as cluster down (YARN-7176) and data loss (HBase-19004). By examining how CloudRaid triggers a bug, we can easily find its root cause and provide a patch. We have provided patches to 6 of the 8 bugs, and 3 of them have been accepted by their original developers.

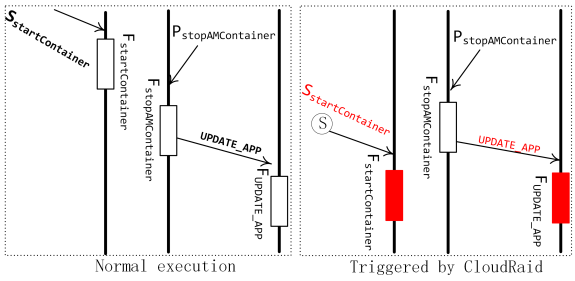

Figure 7: New atomicity violation bug detected by CloudRaid. Messages and their handlers triggering the bug are highlighted in red.

YARN-7176. CloudRaid detects a new atomicity violation bug in Hadoop2/Yarn, as depicted in Figure 7. It successfully flips the normal execution order $S_{\texttt{startAMContainer}} \rightarrowtail P_{\texttt{stopAMContainer}}$, so that the message $P_{\texttt{stopAMContainer}}$ is handled first. The message handler $F_{\texttt{stopAMContainer}}$ sends an `UPDATE_APP` message. As a result, its message handler $F_{\texttt{UPDATE\_APP}}$ executes concurrently with the handler $F_{\texttt{startContainer}}$. A race condition is triggered by their concurrent execution and an `ArrayIndexOutBoundsException` is thrown, crashing the Yarn daemon process.

YARN-7663. In YARN-7663, CloudRaid triggered an `InvalidStateTransitionException` error after reordering the `START` message with the `KILL` message. Hence, in our initial patch, we simply ignore the `START` message if it arrives after the `KILL` message. The original developer accepted our fix, saying "*Ignoring the START event seems to be appropriate here*". However, he made another request "*Could you add a unit test of the new start-after-killed transition logic*"? We then prepared our second patch with a unit test. Interestingly, the unit test triggered another two bugs (YARN-7726 and YARN-7703), both are similar `InvalidStateTransitionException` errors in the state machine implementation of YARN. Although the developers of YARN have tested the state machine implementation with a large set of unit tests, there are still numerous subtle cases not handled. After another 4 different versions of patches (2 months after we reported the bug), the developer finally accepted our patch and submitted it to the latest trunk and some previous trunks (branch-2, branch 2.8, and branch 2.9).

Discussion. The effectiveness of CloudRaid largely relies on the log quality of the system under testing. For systems with rich log information (Hadoop2/Yarn and HBase), it is very effective. However, if the system only provides limited logs (Cassandra), its ability is largely restricted.

4.3 RQ3: Efficiency

Table 8: Analysis and testing times of CloudRaid.

System	Profiling(s)	Analysis(s)	Trigger(s)
Hadoop2/Yarn	648.0	131.3	6990.2
HDFS	646.0	60.0	828.3
HBase	1309.0	63.3	1368.0
Cassandra	263.1	112.3	60.3

Table 8 reports the times in testing the latest versions of different systems with CloudRaid. Column 2 is the time in profiling each system, i.e., running each workload 20 times. In practice, we can get logs from live systems without profiling. Column 3 is the total analysis time, including the time to analyze the source code, and the time to parse all runtime logs from the 20 runs. Column 4 is the testing time to trigger all selected orderings.

CloudRaid is very efficient. It finishes its analyses in 2 minutes for all benchmarks (Column 3). In the testing phase (Column 4), CloudRaid finishes testing Hadoop2/Yarn in 6990.2 second (1.94 hours). Cassandra takes less than 1 minute to test. It is because CloudRaid can only extract very limited information from its runtime logs (Table 2), resulting in only 4 message orderings to be tested.

Table 9: Message orderings pruned by each analysis. #Total is the number of messages orderings. HB is the percentage of orderings pruned by HB analysis. Order is the percentage of orderings already exercised. ID is the percentage of orderings where messages do not log related ID values.

System	#Total	% of Pruned			
		HB	Order	ID	All
Hadoop2/Yarn	4489	1.0%	11.1%	81.5%	93.6%
HDFS	81	2.5%	45.7%	51.9%	85.2%
HBase	324	2.5%	57.7%	34.3%	94.4%
Cassandra	16	0.0%	75.0%	0.0%	75%

Table 9 shows how CloudRaid achieves efficiency by pruning message orderings using different analyses. Note that here we already filtered out these static messages not logged with ID values (Column 2). Otherwise, the total number of message orderings to be tested is 154,449 for Hadoop2/Yarn. Overall, CloudRaid successfully prunes 93.6% of total message orderings for Hadoop2/Yarn, and 94.4% for HBase (Column 6). HB analysis only prunes very few message orders. It is very difficult to precisely analyze the happen before order statically, due to the complexity in these systems. CloudRaid efficiently prunes most message orderings by skipping those that have already been exercised (Column 4), and those between unrelated messages (Column 5). We randomly tested 50 pruned message orderings, and cannot find any new bugs. This confirms our observation and assumption: messages orderings pruned away by CloudRaid are unlikely to expose errors.

When CloudRaid tries to exercise all message orderings, we find that only 24.6% of them are triggered. For these message orders not exercised, 82.1% of them do have strict happen before relation but our HB analysis fails to analyze the happen before order due to unrecognized control or data dependencies. Hence, a more sophisticated may-happen-in-parallel analysis [53] can further improve efficiency. The other 17.9 % of them are due to the fact that our current workload does not cover the specific ordering.

Discussion. CloudRaid drastically improves efficiency by pruning away message orderings that are unlikely to expose errors. We manually tested these message orderings pruned by CloudRaid, and can verify that they do not expose errors. Nevertheless, it trades soundness for efficiency and we cannot guarantee that the pruned message order will not trigger any error.

5 RELATED WORK

We summarize previous works in detecting distributed concurrency bugs, and existing log analysis techniques.

Distributed concurrency bug detection. There is a large body of research on distributed system model checkers [19, 27, 29, 33, 41]. These systems intercept messages in the system at runtime, then permute their orderings exhaustively. Although powerful, they face the state-space explosion problem. Recent tools [19, 29] have adopted various state reduction techniques, to address this problem. However, the more events included, the larger the state space to be explored. It takes up to days to explore some of the state space [29].

Liu et al. [34] recently extended classic race detection techniques for multi-threaded programs [11, 21, 25, 37, 38, 40], to detect race

conditions in distributed systems. Their technique instruments memory accesses and communication events in the target system to collect runtime traces during execution. An offline analysis is performed to analyze the happen-before relation between memory accesses, using a sophisticate happen-before model customized to distributed systems. Concurrent memory accesses that may trigger exceptions are regarded as harmful data races. A trigger is employed to further verify the detected race conditions. CloudRaid differs from their approach in that we mine logs to recover runtime traces, without instrumentation. In addition, we target errors triggered by reordering messages, as those distributed system model checkers, instead of memory accesses only.

Fault injection techniques [8, 14, 16, 17, 22–24, 43, 46] are commonly used to test the resilience of distributed systems. Existing techniques focus on how to inject fault at different system state, to expose bugs in the fault recover handler. CloudRaid can be applied together with these techniques, to detect fault-related concurrency bugs more effectively.

Log Analysis. Many researchers [3, 4, 6, 9, 12, 26, 35, 36, 43, 44, 49] mine logs to extract various information, including temporal invariants [3, 4], user request flow [36, 49], system architecture [35], timing information [6], etc. The mined information can then be applied to help with better understanding, monitoring, and analyzing the complicated distributed systems.

Xu et al. [48] mine console logs from a system and apply machine learning techniques to detect anomaly executions. Mined information such as logged values and logging frequencies are visualized to help user diagnose anomaly behavior. DISTALYZER [39] compares logs from abnormal execution and normal execution to infer the strongest association between system components and performance. Iprof [52] extracts request ID and timing information from logs to profile request latency. Stitch [51] organizes log instances into tasks and sub-tasks, by analyzing relations between logged ID variables, to profile different components in the entire distributed software stack. CloudRaid mines logs to uncover insufficiently exercised message orderings, to effective detect new concurrency bugs.

6 CONCLUSION

We present CloudRaid, a simple yet effective tool to detect distributed concurrency bugs. CloudRaid achieves efficiency and effectiveness at the same time by analyzing message orderings that are likely to expose errors from existing logs. Our evaluation shows that CloudRaid is easy to be adopted and is very effective in detecting bugs. It finishes testing 40 versions of 4 different systems in 35 hours, and have successfully found 28 bugs, including 8 new bugs that have never been reported before.

ACKNOWLEDGEMENT

This work is supported by the National Key research and development program of China (2016YFB1000402), the Innovation Research Group of National Natural Science Foundation of China (61521092 and 61672492), and the National Natural Science Foundation of China (U1736208).

REFERENCES

[1] 2018. Google Protocol Buffer. (2018). Retrieved April 26, 2018 from https://developers.google.com/protocol-buffers/.
[2] 2018. WALA Home page. (2018). Retrieved April 26, 2018 from http://wala.sourceforge.net/wiki/index.php/Main_Page/.
[3] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, Arvind Krishnamurthy, and Thomas E Anderson. 2012. Mining temporal invariants from partially ordered logs. *ACM SIGOPS Operating Systems Review* 45, 3 (2012), 39–46.
[4] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 267–277.
[5] Dhruba Borthakur et al. 2008. HDFS architecture guide. *Hadoop Apache Project* 53 (2008).
[6] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services.. In *OSDI*. 217–231.
[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492
[8] Florin Dinu and TS Ng. 2012. Understanding the effects and implications of compute node related failures in hadoop. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 187–198.
[9] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1285–1298.
[10] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 78, 12 pages. http://dl.acm.org/citation.cfm?id=2388996.2389102
[11] Cormac Flanagan and Stephen N Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, Vol. 44. ACM, 121–133.
[12] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 149–158.
[13] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.
[14] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*.
[15] Lars George. 2011. *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.".
[16] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of NSDIâĂŹ11: 8th USENIX Symposium on Networked Systems Design and Implementation*. 239.
[17] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Koushik Sen. 2010. Towards Automatically Checking Thousands of Failures with Micro-specifications.. In *HotDep*.
[18] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 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 '14)*. ACM, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/2670979.2670986
[19] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 265–278.
[20] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Peter Bodik, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. 2013. Failure Recovery: When the Cure is Worse Than the Disease. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=2490483.2490491
[21] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L Pereira, Gilles A Pokam, Peter M Chen, and Jason Flinn. 2014. Race detection for event-driven mobile applications. *ACM SIGPLAN Notices* 49, 6 (2014), 326–336.

[22] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. 2013. SETSUDŌ: perturbation-based testing framework for scalable distributed systems. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM, 7.

[23] Pallavi Joshi, Haryadi S Gunawi, and Koushik Sen. 2011. PREFAIL: A programmable tool for multiple-failure injection. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 171–188.

[24] Xiaoen Ju, Livio Soares, Kang G Shin, Kyung Dong Ryu, and Dilma Da Silva. 2013. On fault resilience of OpenStack. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2.

[25] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data races vs. data race bugs: telling the difference with portend. *ACM SIGPLAN Notices* 47, 4 (2012), 185–198.

[26] Kamal Kc and Xiaohui Gu. 2011. ELT: Efficient log-based troubleshooting system for cloud computing infrastructures. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE, 11–20.

[27] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, death, and the critical transition: Finding liveness bugs in systems code. NSDI.

[28] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[29] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems.. In *OSDI*. 399–414.

[30] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

[31] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-sensitive Points-to Analysis Using Value Flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 343–353. https://doi.org/10.1145/2025113.2025160

[32] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and Scalable Context-sensitive Pointer Analysis via Value Flow Graph. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/2464157.2466483

[33] Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent model checking of unmodified distributed systems. In *6th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*.

[34] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 677–691.

[35] Jian-Guang Lou, Qiang Fu, Yi Wang, and Jiang Li. 2010. Mining dependency in distributed systems through unstructured logs analysis. *ACM SIGOPS Operating Systems Review* 44, 1 (2010), 91–96.

[36] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Jiang Li, and Bin Wu. 2010. Mining program workflow from interleaved traces. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 613–622.

[37] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: detecting atomicity violations via access interleaving invariants. In *ACM SIGOPS Operating Systems Review*, Vol. 40. ACM, 37–48.

[38] Brandon Lucia, Luis Ceze, and Karin Strauss. 2010. ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. *ACM SIGARCH computer architecture news* 38, 3 (2010), 222–233.

[39] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 26–26.

[40] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

[41] Jiri Simsa, Randal E Bryant, and Garth Gibson. 2010. dBug: systematic evaluation of distributed systems. USENIX.

[42] Yulei Sui and Jingling Xue. 2016. On-demand Strong Update Analysis via Value-flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 460–473. https://doi.org/10.1145/2950290.2950296

[43] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. 2010. Visual, log-based causal tracing for performance debugging of mapreduce systems. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*. IEEE, 795–806.

[44] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. 2008. SALSA: Analyzing Logs as StAte Machines. *WASL* 8 (2008), 6–6.

[45] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 278–291. https://doi.org/10.1145/3062341.3062360

[46] Hadoop Team. 2018. Fault Injection framework. (2018).

[47] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[48] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 117–132.

[49] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. 2016. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 489–502.

[50] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 249–265. http://dl.acm.org/citation.cfm?id=2685048.2685068

[51] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle.. In *OSDI*. 603–618.

[52] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems.. In *OSDI*, Vol. 14. 629–644.

[53] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. 2018. May-happen-in-parallel Analysis with Static Vector Clocks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 228–240. https://doi.org/10.1145/3168813