CloudRaid: Detecting Distributed Concurrency Bugs via Log Mining and Enhancement

Jie Lu, Feng Li, Chen Liu, Lian Li, Xiaobing Feng and Jingling Xue

Abstract—Cloud systems suffer from distributed concurrency bugs, which often lead to data loss and service outage. This paper presents CLOUDRAID, a new automatical tool for finding distributed concurrency bugs efficiently and effectively. Distributed concurrency bugs are notoriously difficult to find as they are triggered by untimely interaction among nodes, i.e., unexpected message orderings. To detect concurrency bugs in cloud systems efficiently and effectively, CLOUDRAID analyzes and tests automatically only the message orderings that are likely to expose errors. Specifically, CLOUDRAID mines the logs from previous executions to uncover the message orderings that are feasible but inadequately tested. In addition, we also propose a log enhancing technique to introduce new logs automatically in the system being tested. These extra logs added improve further the effectiveness of CLOUDRAID without introducing any noticeable performance overhead. Our log-based approach makes it well-suited for live systems.
We have applied CLOUDRAID to analyze six representative distributed systems: Hadoop2/Yarn, HBase, HDFS, Cassandra, Zookeeper, and Flink. CLOUDRAID has succeeded in testing 60 different versions of these six systems (10 versions per system) in 35 hours, uncovering 31 concurrency bugs, including nine new bugs that have never been reported before. For these nine new bugs detected, which have all been confirmed by their original developers, three are critical and have already been fixed.

Index Terms—Distributed Systems, Concurrency Bugs, Bug Detection, Cloud Computing.

1 INTRODUCTION

Distributed systems, such as scale-out computing frameworks [1], [2], distributed key-value stores [3], [4], scalable file systems [3], [4] and cluster management services [2], are the fundamental building blocks of modern cloud applications. As cloud applications provide 24/7 online services to users, high reliability of their underlying distributed systems becomes crucial. However, distributed systems are notoriously difficult to get right. There are widely existing software bugs in real-world distributed systems, which often cause data loss and cloud outage, costing service providers millions of dollars per outrage [5], [6].

Among all types of bugs in distributed systems, distributed concurrency bugs are among the most troublesome [7], [8]. These bugs are triggered by complex interleavings of messages, i.e., unexpected orderings of communication events. It is difficult for programmers to correctly reason about and handle concurrent executions on multiple machines. This fact has motivated a large body of research on distributed system model checkers [9], [10], [11], [12], which detect hard-to-find bugs by exercising all possible message orderings systematically. Theoretically, these model checkers can guarantee reliability when running the same workload verified earlier. However, distributed system model checkers face the state-space explosion problem [9]. Despite recent advances [9], it is still difficult to scale them to many large real-world applications. For example, in our experiments for running the WordCount workload on Hadoop2/Yarn, 5,495 messages are involved. Even in such a simple case, it becomes impractical to test exhaustively all possible message orderings in a timely manner.

This paper proposes a novel strategy for detecting distributed concurrency bugs. Instead of trying all possible message orderings exhaustively, we test selectively only those message orderings that are likely to expose bugs. Which message orderings are likely to trigger errors then? We address this key question based on two observations:

> **Observation1.** In bug-triggering message orderings, their corresponding message handlers are *access-related*, i.e., access shared resources. Errors can be triggered when the shared resources are accessed inconsistently under different message orderings.

> **Observation2.** 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. Using these tested message orderings to expose errors is unlikely to be beneficial.

1.1 The Approach

Therefore, we introduce a log-based approach that learns from previous executions to uncover the message orderings

- *Jie Lu, Chen Liu, Lian Li, and Xiaobing Feng are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology of the Chinese Academy of Sciences China and University of Chinese Academy of Sciences,Beijing 100049, China. Feng Li is with the Key Laboratory of Network Assessment Technology, Institute of Information Engineering of Chinese Academy of Sciences China and Beijing Key Laboratory of Network Security and Protection Technology, China. Jingling Xue is with University of New South Wales, Sydney, Australia. Lian Li and Feng Li are corresponding authors.*
  E-mail: {lujie,lianli,liuchen17z,fxb}@ict.ac.cn, lifeng@iie.ac.cn, jingling@cse.unsw.edu.au.

that are likely to expose errors, i.e., access-related message orderings that are feasible but not yet tested. Since we harness the rich execution history from live systems, a non-intrusive log-based approach is thus desirable. Modern cloud applications often provide a rich set of runtime logs, which record important events, to help with the diagnosis and monitoring of online systems. Our log-based approach makes it well-suited for live systems, for which intrusive instrumentation is often not an option.

According to a previous study [8], more than 60% distributed concurrency bugs can be triggered by a single untimely message delivery. Therefore, our approach focuses on examining the orderings between a pair of messages. Errors involving more than two messages can be handled by considering multiple pairs of messages together as follows:

- Starting from a pair of access-related messages $< S, P >$, where S always arrives before P, i.e., $S \nrightarrow P$ (according to existing logs), our approach tries to flip the order so that $P \nrightarrow S$ can be also exercised, provided that both messages may happen in parallel.
- For two pairs $< S, P >$ and $< P, Q >$, we exercise first $P \nrightarrow S$ and $Q \nrightarrow P$ individually and then $P \nrightarrow S$ and $Q \nrightarrow P$ together. Thus, all permutations of the three messages are tried. Similarly, message sequences involving more messages are handled.

Our approach focuses on detecting the bugs caused by order violation, i.e., the bugs which manifest themselves whenever a message arrives at a wrong order with respect to another event. The majority of these bugs can be exposed by reordering a pair of messages, as suggested previously [8]. However, relatively few but critical bugs still occur when more than two messages are involved. These bugs can only be exposed under special timing conditions, involving, for example, some specific messages or events (e.g., node crashes or reboots). To detect such errors, we have empowered our approach with the capability of reordering an arbitrary number of messages for an application.

1.2 The Tool

We have designed and implemented CLOUDRAID, a new tool for detecting distributed concurrency bugs efficiently and effectively. CLOUDRAID extracts automatically sequences of important communication events from existing run-time logs, enriched by also a new log enhancing technique. Only permutations of these access-related message orderings will be tested, provided that they are feasible but not yet exercised. A dynamic trigger is employed to exercise the selected message orderings at runtime.

We have applied CLOUDRAID to six representative distributed systems: Hadoop2/Yarn [2], HDFS [13], HBase [3], Cassandra [4], Zookeeper [14], and Flink [15]. CLOUDRAID is simple to deploy as the system under testing first runs without any modification. Then in a separate testing phase, our dynamic trigger performs minimal instrumentation to test a specific message ordering. In our evaluation, we have chosen randomly 60 different versions of these systems (10 versions per system) and run a total of six different workloads on these systems. CLOUDRAID runs for 3200 times in 35 hours, where each run tries to exercise a specific message ordering. We have successfully triggered 31 bugs (with no false positives), including nine new bugs that have never been found before. All the nine new bugs have been confirmed by the original developers, with three of them considered as critical bugs and thus already fixed.

This paper makes the following contributions:

- We propose a new approach, CLOUDRAID, for detecting concurrency bugs in distributed systems efficiently and effectively. CLOUDRAID leverages the run-time logs of live systems and avoids unnecessary repetitive tests, thereby drastically improving the efficiency and effectiveness of our approach.
- We describe a new log enhancing technique for improving log quality automatically. This enables us to log key communication events in a system automatically without introducing any noticeable performance penalty. The enhanced logs can further improve the overall effectiveness of our approach.
- We have evaluated extensively CLOUDRAID using six representative distributed systems: Hadoop2/Yarn, HBase, HDFS, Cassandra, Zookeeper, and Flink. CLOUDRAID can test 60 different versions of these six systems (with six workloads in total) in 35 hours, and detect successfully 31 concurrency bugs. Among them, there are nine new bugs, including three critical ones, which have 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. Section 3 presents the design and implementation of CLOUDRAID. Section 4 evaluates its efficiency and effectiveness. Section 5 reviews the related work. Finally, Section 6 concludes.

2 A MOTIVATING EXAMPLE

Figure 1 gives an example that illustrates a typical scenario for creating a new task in Hadoop MapReduce. There is a concurrency bug known as MAPREDUCE-3656.

The Bug. The two messages, (5) and (9), that trigger the MAPREDUCE-3656 bug, together with their corresponding event handlers, are highlighted in red. During the normal execution, the remote procedure call (RPC) to `startContainer` finishes its execution quickly. Hence, the `ASSIGNED` event (i.e., message (5)) is always dispatched and handled before the `UPDATE` event (i.e., message (9)). However, there is no happen-before relation in between. If the `UPDATE` event arrives before the `ASSIGNED` event (due to an unexpected delay in T_1, e.g., insufficient resources in 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, leading to job fail. The fix is to introduce smart synchronizations to guarantee that the `ASSIGNED` event always arrives before the `UPDATE` event.

Among all the messages for this example, the message pair <(5), (9)> is the root cause of the error. Both messages are handled by the same event handling method to update the same variable for the task status. How can we select automatically the message order (9) $\nrightarrow$ (5) among all the messages? Let us dig into some technical details below.

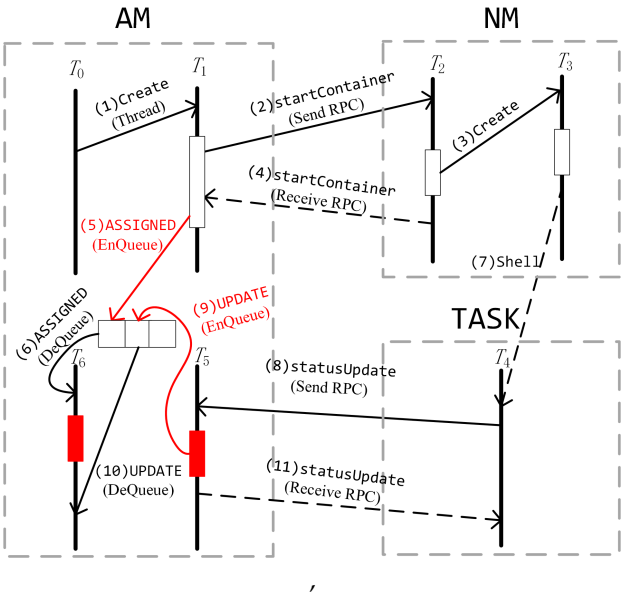

Fig. 1. A real-world example for starting a new task in Hadoop2, with the messages (5) and (9) for triggering the MAPREDUCE-3656 bug and their corresponding event handlers being highlighted in red. AM is an application manager node, NM is a node manager node, and TASK is the node for running the task. (1) Thread T_0 in AM creates a new thread T_1. (2) T_1 invokes the remote procedure `StartContainer` to start a container on NM (Thread T_2). (3) In `StartContainer`, T_2 creates another thread T_3. (4) The RPC to `StartContainer` returns to T_1. (5) After returning from `StartContainer`, T_1 sends an `ASSIGNED` event to the event queue. (6) The `ASSIGNED` event is dispatched to the event handler T_6, with the TASK state updated to `ASSIGNED`. (7) T_3 starts a new process to run the task on node TASK using a shell script (Thread T_4). (8) T_4 invokes the remote procedure `statusUpdate` on AM (Thread T_5). (9) In T_5, `statusUpdate` sends an `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) The RPC to `statusUpdate` returns to T_4.

2.1 Source Code and Runtime Logs

Figure 2 gives a code snippet abstracted for our example in Figure 1. The lines for sending messages are highlighted in red. We call those source locations for sending messages *static messages*. Hereafter, we write M to denote a static message and M_i for one of its dynamic instances.

Static messages manifest themselves in three common patterns, thread creation (message (1) in line 3), RPC (message (2) in line 11), and event dispatch (message (5) in line 12 and message (9) in line 38). For simplicity, the code for sending messages (3) and (7) is elided. The event handling method `EventHandler.handle` (line 19) calls `StateMachine.doTransition`, which invokes different callback functions to handle different types of events. Here we present a simplified version with callbacks inlined.

The lines highlighted in bold log static messages. All messages, except for the RPC return (messages (4) and (11)) and the call to the 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 `statusUpdate` in line 36). Hence, a common log (line 37) is introduced to serve both messages for performance reasons. In this case, we group the two static messages together, denoted as (8,9).

A log consists of constant strings and values of variables. The code snippet in Figure 2 will execute multiple times at run time, resulting in multiple dynamic instances per static message and multiple log instances. The values of variables in the log instances are used to distinguish each dynamic instance. Figure 3 shows the two simplified runtime logs for two different executions of this code snippet.

2.2 Methodology

Ideally, we would like to recover precisely 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). The log instances from the same run are grouped together in order. In reality, we perform source code analysis and log analysis together to infer such message sequences. We analyze how static messages are handled and logged. Runtime log instances can then be mapped to static messages with static analysis information. We group the logs from the same run together by analyzing the relations among the logged variable values (ID 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. We focus on the orderings for pairs of access-related static messages $< P, S >$, where P and S may happen in parallel. As discussed in Section 1, message sequences involving more than two messages are tested by considering multiple such message pairs together. Some message pairs follow a strict happen-before order, e.g., <(1),(2)>, <(2),(3)>, and <(2),(5)>. The underlying order cannot be mutated. Our two observations stated in Section 1 provide the basic guidelines to select a message ordering $P \nrightarrow S$ as follows:

> **Rule 1:** Runtime log instances P_i and S_i must log all *access-related* ID variables, and
> **Rule 2:** The order $P_i \nrightarrow S_i$ has not been exercised.
>
> In both rules, P_i and S_i are runtime instances of P and S with matching values of ID variables.

Observation 1 leads to Rule 1. Distributed systems frequently use the values of ID variables as indexes to access shared resources. Thus, messages logging completely unrelated values of ID variables are unlikely to access commonly shared objects (i.e., not access-related), and unlikely to expose errors. Rule 2 discards those 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 the values of related ID variables (`containerID` and `taskAttemptID`) in their logs (Rule 1). The message orderings (3) $\nrightarrow$ (5), 5 $\nrightarrow$ (3) and (5) $\nrightarrow$ (8,9) have already been tested, according to the log information. Hence, we will select the order (8,9) $\nrightarrow$ (5) for further testing. The error can then be triggered.

3 The CloudRaid Approach

We have designed and implemented CloudRaid in WALA [16] via a series of sub-analyses (Figure 4):

```
// Thread T_0 in AM
public void ContainerLauncherImpl.serviceStart() {
    Runnable t = createEventProcessor(new ContainerLaucherEvent());
    this.launcherPool.execute(t);  //  Message (1)
}
public void ContainerLauncherImpl.createEventProcessor(ContainerLauncherEvent event) {
    return new EventProcessor(event);
}

// Thread T_1 in AM, handler of Message(1)
public void EventProcessor.run() {
    LOG.info("Launching " + this.taskAttemptID);
    ContainerManagementProtocolPBClientImpl proxy = getCMProxy(this.containerMgrAddress);
    StartContainerResponse response = proxy.startContainer(new StartConReq(...)); // Message (2)
     this.dispatcher.handle(new TaskAttemptContainerLaunchedEvent(this.taskAttemptID...));   // Message (5)
}

// Thread T_2 in NM, handler of Message(2)
public StartConRes ContainerManagerImpl.startContainer(StartConReq req) {
    ID containerID = req.getConLauContext().getConId();
    LOG.info("Start request for " + containerID);
    ... // create thread T_3 via thread pool, message(3)
}

// Thread T_6 in AM, handler of Message(5) and Message(9) (dispatched by the event dispatcher)
public void EventHandler.handle(TaskEvent event){
    TaskTAttemptEvent ev = (TaskTAttemptEvent) event;
    if (this.oldState== ASSIGNED&& ev.getType().equals(TA_CONTAINER_LAUNCHED)) {
        // Handle message (5), callback1
        TaskAttempt attempt = ev.getTaskAttempt();
        LOG.info("TaskAttempt: ["+attempt.attemptId+"] using containerId: ["+attempt.containerID);
        . . .  // Update task status
    } else if (this.oldState== ASSIGNED && ev.getType().equals(TA_CONTAINER_UPDATE)) {
        // Handle message (9), callback2
        . . .
    }   // Other cases
}

// Thread T_3 in NM, handler of Message(3)
public void LocalizerRunner.run() {
    nmPrivateCTokensPath = getLocalPathForWrite(this.localizerId);
    LOG.info("Writing credentials to the nmPrivate file " + nmPrivateCTokensPath.toString());
    . . . // create TASK via Shell script, message (7)
}

// Thread T_5 in AM, handler of Message(8)
public void TaskAttemptListenerImpl.statusUpdate(TaskAttemptID taskAttemptID) {
     LOG.info("Status update from " + taskAttemptID);
     this.dispatcher.handle(new TaskAttemptStatusUpdateEvent(. . .)); // Message (9)
}
```

Fig. 2. Code snippet for implementing the scenario illustrated in Figure 1.

```
Launching attempt_..._0001_m_000009_0    // Message (1)
Start request for container_1514878932605_0001_01_000011 // Message (2)
Writing credentials to the nmPrivate file
$HADOOP_HOME/nm-local-dir/nmPrivate/container_1514878932605_0001_01_000011.tokens // Message (3)
TaskAttempt: [attempt_..._0001_m_000009_0] using containerId: [container_1514878932605_0001_01_000011] //Message (5)
Status update from attempt_1514878932605_0001_m_000009_0 // Message (8), immediately followed by (9)

Launching attempt_..._0002_m_000007_0 // Message (1)
Start request for container_1514878932605_0002_01_000009 // Message (2)
TaskAttempt: [attempt_..._0002_m_000007_0] using containerId: [container_1514878932605_0002_01_000009] //Message (5)
Writing credentials to the nmPrivate file
$HADOOP_HOME/nm-local-dir/nmPrivate/container_1514878932605_0002_01_000009.tokens // Message (3)
Status update from attempt_1514878932605_0002_m_000007_0 // Message (8), immediately followed by (9)
```

Fig. 3. Simplified runtime logs for two different executions of the code snippet in Figure 2.

- *Communication Analysis* analyzes statically how static messages are handled and logged. Each static message is annotated with a logging expression. Proceeding similarly as in [17], [18], we represent logging expressions of static messages as regular expressions.
- *Log Analysis* uses the logging expressions to map each runtime log instance to a static message.
- *ID Analysis* analyzes the relations among the logged values, according to the dependences between logged variables and their run time values (from the log analysis). The messages in the same run can then be distinguished from those in the other runs and grouped together (as discussed in Section 2).
- *Log Enhancer* is an optional component for introducing automatically new logging statements into the system being tested to improve log quality.
- *HB Analysis* analyzes statically the happen-before order among the static messages.
- *Message Pair Analysis* explores message orderings to perform further tests, based on the results obtained

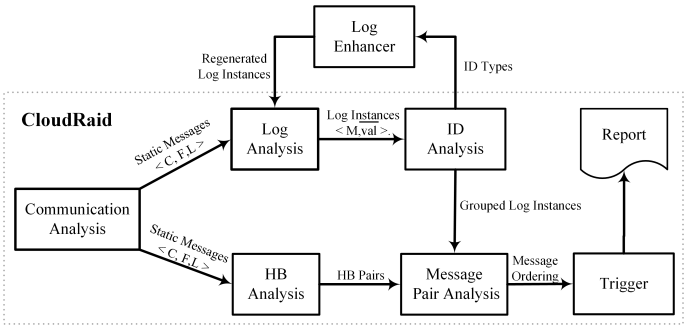

Fig. 4. Architecture of CLOUDRAID.

by the HB and ID analyses.

- *Trigger* will instrument the source code to exercise the selected message orderings (to expose bugs).

3.1 Communication Analysis

In CLOUDRAID, communication analysis is the basis for all subsequent analyses. Each static message is represented as a triple $< C, F, L >$, where C is the client site for sending the message, F is the corresponding message handler, and L is its logging expression. Table 1 lists the static messages in our motivating example (with a client site identified by its source line number).

Below we first describe how to handle three common patterns used for static messages, thread creation, RPC, and event dispatch, and then summarize our analysis.

3.1.1 Thread Creation

The client site C is a call site to one of the two methods: `t.start()` and `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 slice the program backwards to find the object that `t` points to. Instead of using a sophisticated pointer analysis [19], [20], [21], [22], we found it sufficient to 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` backwards, 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 expression 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 a local call site, and the message handler F is the invoked remote procedure. In common practices (e.g., google protobuf [23]), RPC is realized via a client class and a corresponding server class, with both implementing the same interface. To identify RPC, we require the user to specify the common interface implemented by RPC client classes and server classes.

Figure 5 shows how to specify RPC and event messages for our motivating example. In common practices, RPC client methods serialize an object, which will then be de-serialized by their corresponding server methods. Hence, we distinguish RPC client classes from RPC server classes by checking whether their implemented interface methods serialize/de-serialize a formal parameter or not. A RPC server class can then be matched with its corresponding RPC client class by checking their public APIs. We recognize automatically classes that wrap RPC client classes using the delegation design pattern [24]. Thus, given a RPC client site, we can find easily its RPC server class and the corresponding remote procedure to handle this message.

```
<RPC>
<Interface>ContainerManager.BlockingInterface</Interface>
</RPC>

<Event>
<Client>Dispatcher.handle
    <Handler>Callback1</Handler>
    <Handler>Callback2</Handler>
</Client>
</Event>
```

Fig. 5. Specifying RPC and event messages for the code in Figure 2.

In Figure 2, class `ContainerManagementProtocolPBClientImpl` is a RPC client class. The call to method `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 identified as <11, `ContainerManagerImpl.startContainer`, L>.

3.1.3 Event Dispatch

The client site C is a call site for enqueuing an event and the message handler F is the method for handling 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 for enqueuing and handling an event.

Figure 5 gives the specification for event messages, where `client` is the method for enqueuing an event and `Handler` represents its corresponding handlers. The client method may invoke different handlers (using callback functions) for distinct types of events, resulting in distinct static messages. Hence, we allow users to specify all potential handlers for an event. As illustrated in Figure 5, a callsite for `Dispatcher.handle` is the client site to enqueue an event, which will be handled by one of the two handlers `Callback1` and `Callback2`. Thus, at each client side C, there will be two static messages <C, `Callback1`, L> and <C, `Callback2`, L>. In most cases, we can associate a unique handler with each client side by checking the type of the enqueued event, i.e., matching the types of enqueued events with those of dispatched events (the formal arguments of the corresponding event handler).

In Figure 2, `Dispatcher.handle` is the method for enqueuing an event and `EventHandler.handle` (line 19) is the method for handling a dispatched event. In actuality, the event handler implements a state machine that invokes different callback functions (i.e., `callback1` and `callback2`) by a case analysis according to the incoming event type

TABLE 1
Static messages < C, F, L > in the code snippet of Figure 2.

Message	Client Site C	Message Handler F	Logging Expression 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_*

TABLE 2
Log instances < $M, \overline{val}$ > for the runtime logs given in Figure 3.

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`

and the current state (method `EventHandler.handle` (line 19) in Figure 2). Here, there are two events, message (5) and message (9). Their client sites are the call sites to `Dispatcher.handle` at lines 12 and 38, respectively. By checking the types of enqueued events, we can deduce that the two messages are handled by `callback1` and `callback2`, respectively (lines 22 and 27).

3.1.4 Message Logging

We first locate a log point (e.g., a call to `Log.info` or one of its 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, the log point for message (2) (<11, `ContainerManagerImpl.start-Container`, L>) is line 16, and the log point for message (5) (<12, `callback1`, L>) is line 22. Hence, we search the entry block of the message handler F and 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 single log point.

However, there is no log point in its handler for message (9). In this case, we will search for a log point at the client site C, i.e., in the basic block containing C. Hence, line 37 is regarded as the log point of message (9). The messages that share the same log point are grouped together. In our example, messages (8) and (9) are grouped together.

The message logging expression at each log point can then be extracted. Following previous work [17], [18], we represent a logging expression in terms of a regular expression. We analyze the logging statement at each log point and the `tostring` method of a logged variable to extract statically the constant strings in the log message. The runtime values of a logged variable are denoted as *. Table 1 gives the regular expressions for all the logging expressions in the code snippet shown in Figure 2.

3.1.5 Discussion

Our communication analysis has been designed to require minimal user specification. For RPC, we require the user to specify the RPC interfaces implemented. For event dispatch, we require the user to specify the event enqueue method and the event handlers for different types of events. The communication analysis can then analyze automatically each message client site, identify its corresponding handler, and locate the right log point to extract its logging expression. Its precision can be further improved if the user can provide detailed annotations to specify the client site, the message handler, and the logging expression for each message. Alternatively, we could obtain such precise information via instrumentation and profiling. For the six different distributed systems studied (Section 4), our communication analysis can correctly extract the logging expressions for the majority of messages without any loss of precision.

3.2 Log Analysis

Log analysis tries to match each runtime log instance with a message logging expression. A log instance is represented as a tuple < $M, \overline{Val}$ >, where M is the static message and $\overline{Val}$ records a list of the runtime values of the logged variables. Table 2 gives the representation of each runtime log instance in Figure 3. Logs are ordered according to their time stamps, which are calibrated to a centralized time to compensate for the time differences on distinct nodes, as in [25].

We adopt the approach in [17] to match each runtime log instance with a static message logging expression efficiently. A reverse index is built as a hash for each logging expression, which can be used to calculate quickly 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 expressions with the highest scores and then parse each log instance according to the 10 logging expressions in order to find an exact match. For each log instance, we also record the runtime values of its logged variables, as shown in Table 2.

```
1  //client
2  public void submitApplication(AppId appId){
3      appContext.setApplicationId(appId);
4      request.setApplicationSubmissionContext(appContext);
5      rmClient.submitApplication(request);
6  }
7  //server
8  public SubmitApplicationResponse submitApplication(
9                    SubmitAppRequest request){
10     Application submissionContext = request.getContext();
11     ApplicationId appId = submissionContext.getAppId();
12     ...
13     LOG.info("Application" + appId + " is submitted");
14     ...
15     return RecordFactory.newRecordInstance();
16 }
```

Fig. 6. Identifying ID Variables for a code snippet from Hadoop2/Yarn.

3.3 ID Analysis

ID analysis organizes all related log instances into a hierarchical structure, based on the *ID Values*, i.e., runtime values of *ID variables*. The hierarchical structure captures the relations among tasks and their sub-tasks. In distributed systems, the 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 it is propagated from a message and printed in logs.

Definition 1. A variable is referred to as an *ID variable* if (1) it is accessible from a formal argument of a message handler or a field of a runnable object (via direct or indirect field dereferencing) and (2) it is printed in logs.

Figure 6 gives an example from Hadoop2/Yarn, where all variables propagated from messages are highlighted in blue. Here, `submitApplication` sets `appID` as a field of a request object, `request` (lines 3 and 4), and then sends this RPC request at line 5. The RPC handler `submitApplication` (lines 8 - 16) decomposes `request` (its formal parameter) to obtain the stored `appID` (lines 10 and 11), which is then printed in log (line 13). According to Definition 1, `request` is identified as an ID variable.

It is difficult to analyze precisely the propagation of ID variables statically, especially when complicated pointer and field dereferencing operations are involved. So we build statically an initial set of ID variables and then use their runtime values to group the log instances with the same ID value together. These variables, which are propagated from formal arguments of message handlers or runnable object fields to a log point (via direct assignments or field dereferences), are included in the initial set of ID variables.

Definition 2. A variable with the same run-time value as an existing ID variable is an ID variable.

In line 33 of our example (Figure 2), we cannot determine statically whether the logged variable `nmPrivateCTokensPath` is propagated from a formal argument or not, since 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 with an existing ID value (`containerID` logged at line 16), by Definition 2.

Definition 3. ID value V_1 and ID value V_2 are said to be *access-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 a static message M. V_1 is said to be 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 conversely.

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

For our motivating example, the log instances in Table 2 are organized into two groups. Each group consists of two access-related tasks (Definition 3), indexed by the runtime values `attempt_*`, and `container_*`, respectively.

3.4 Log Enhancement

We rely on a new log enhancing technique to improve further the effectiveness of CLOUDRAID. Unlike previous work [26], [27], [28], [29], which focuses on introducing new logs for error-prone code to facilitate debugging, we propose to enrich logging information in order to achieve more precise message recovery and more thorough testing. There are two questions to be addressed:

- Where to log, i.e., where to place logging statements?
- What to log, i.e., which variables need to be logged?

3.4.1 Where to Log?

CLOUDRAID relies on runtime logs to recover message events. All the recovered message events are organized into groups, according to the runtime values of ID variables in their corresponding logs. We adhere to the guideline:

> A static message $< C, F >$ should be logged if there exists a variable V in F such that V is an ID variable.

As a result, the new logs introduced will help CLOUDRAID uncover successfully more message events and group them accordingly. To analyze statically whether a variable is access-related to an existing ID variable or not, we have developed a type-based analysis as follows:

Definition 4. A type T is said to be an *ID-type* if there exists an ID variable V with type T. A variable with an ID-type is regarded as an ID variable.

In this simple type analysis, we skip the four base types, `Integer`, `String`, `File`, and `Byte`, to reduce false positives. Hence, we introduce logging statements for certain static messages if they have no log points and their corresponding message handlers contain local ID variables.

Figure 7 gives an example. Here, `signalToContainer` is an RPC message handler and `containerId` is a local ID variable with an ID-type `ContainerId`. Thus, `containerId` is an ID variable. As a result, a logging statement is introduced after line 3 (Definition 4). On the other hand, there is no local ID variable in the event handler `SignalContainerTransition#transition`. Thus, we will not introduce logs for that message. Intuitively, the local ID variable in the message handler suggests that the handler performs some complex operation, which is likely to introduce errors and needs to be logged.

```
1  //RPC
2  public void signalToContainer(
                SignalContainerRequest request){
3      ContainerId containerId = request.getContainerId();
+      Log.info("signalToContainer Line 3" + containerID);
                // Send event
4      this.rmContext.handle(
                new RMNodeSignalContainerEvent(...));
5      . . .
6  }

7  //Event handled by Callback
8  public void SignalContainerTransition#transition(
                RMNodeEvent event){
9     rmNode.containerToSignal.add(event.getSignalRequest());
10 }
```

Fig. 7. The code snippet that misses a log statement.

3.4.2 What to Log?

Each logging statement prints a unique line number so that its runtime log instance can be mapped easily to a logging statement. The local ID variables are also printed in order to group their log instances accordingly. To avoid printing unnecessary ID variables, we apply the following rule:

> A local ID variable V is printed in logs if there is no other ID variable V' such that (1) V' is a sub-ID of V or (2) V' is both printed in logs and access-related to V.

Thus, a logged ID variable can help CLOUDRAID organize log instances into groups. A simple type-based analysis is employed to determine statically whether or not a variable is a sub-ID of or access-related to another variable.

Definition 5. Let a variable V_1 (V_2) be of type T_1 (T_2). Let a variable V_1' (V_2') be of type T_1 (T_2). Then V_1' is access-related to V_2' if V_1 and V_2 are access-related. In addition, V_1' is a sub-ID of V_2' if V_1 is a sub-ID of V_2.

3.5 HB Analysis

HB analysis infers the happen-before relation among all the static messages. We consider two representative scenarios. Suppose there are two static messages $P :< C_P, F_P, L_P >$ and $S :< C_S, F_S, L_S >$. If C_S resides 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 then compute the transitive happen-before relation for all static messages.

3.6 Message Pair Analysis

Message pair analysis selects the order $P \nrightarrow S$ for two messages P and S for further testing. If P has already happened before S or S can never happen before P, then $P \nrightarrow S$ either always holds or is infeasible, and should thus not be selected. We check whether P and S are related or not and whether $P \nrightarrow S$ has been exercised or not by comparing their log instances in a pair-wise manner.

Consider two log instances $P_i :< P, - >$ and $S_i :< S, - >$. If P_i and S_i belong to the same task or two 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. Therefore, $P \nrightarrow S$ will be selected only when (1) P and S are related and (2) the order has not been exercised yet.

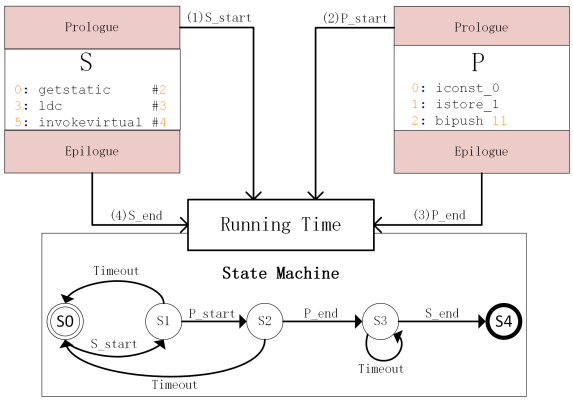

Fig. 8. Triggering order $P \nrightarrow S$ by flipping $S \nrightarrow P$.

3.7 Error Triggering and Reporting

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

3.7.1 Triggering Errors

3.7.1.1 Triggering One Message Order Pair: As shown in Figure 8, we maintain a state machine at run time. Initially, neither P nor S is executed. The start state is `S0`. When executing S, the prologue of S sends an $\texttt{S}_{\texttt{start}}$ 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. T is set to be the largest gap between the related instances of P_i and S_i in previous executions i. We introduce a δ to compensate for the delays caused by our instrumented code and for handling message P. As a result, P will have a good chance to be handled while S is waiting. When P starts to execute, the prologue of P sends a $\texttt{P}_{\texttt{start}}$ event, causing the state machine to advance to `S2`. After P is done, the epilogue of P sends a $\texttt{P}_{\texttt{end}}$ event, reaching state `S3`. After waiting for $T + \delta$, the prologue of S sends a `Timeout` event and then continues its execution. If P has already finished its execution (reaching state `S3`), the message order can be successfully exercised. The state machine reaches the final state `S4` when S finishes its execution. Otherwise, if P does not arrive in time (captured by state `S1`) or has not finished its execution (captured by state `S2`), the runtime state will be reset to its initial state, suggesting that we have not successfully triggered the order as desired.

3.7.1.2 Triggering Multiple Message Order Pairs: CLOUDRAID triggers a message ordering involving three or more messages by flipping the order of multiple message pairs together as described in Section 1. For example, we can

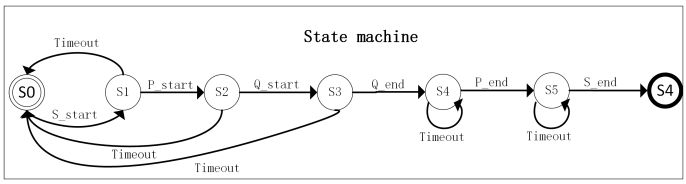

Fig. 9. State machine for three messages.

flip the order of $< S, P >$ and $< P, Q >$ to trigger $Q \rightarrowtail P \rightarrowtail S$. Figure 9 gives the corresponding state machine. S will wait for P to be executed (from state $S1$), which will wait for Q (from state $S2$). The timeout internal for S is set to be the largest gap between S and Q, augmented again by a δ to compensate for the delays caused by instrumented code and for handling messages P and Q.

We use a python script to generate automatically a state machine as needed. Given the number of messages in a message ordering, the script will automatically generate a state machine and all possible message orderings.

3.7.2 Reporting Errors

CLOUDRAID reports an error in three cases: (1) system crash, (2) job hang or fail, and (3) uncommon errors in the log file. Uncommon errors are exceptions related to memory errors, e.g., null pointer exceptions and out-of-bounds exceptions, and exceptions that have been clarified as bugs before, e.g., InvalidStateTransitionException in YARN. Common errors such as network delay errors are elided.

Currently, we do not report silent errors which lead to unexpected behaviors that are difficult to detect, e.g., silent data corruptions [30]. How to develop test oracles to automatically detect such unexpected behaviors is an important topic worth a separate investigation.

4 EVALUATION

We have applied CLOUDRAID to six representative real-world distributed systems: Apache Hadoop2/Yarn (a distributed computing framework), HDFS (a distributed file system), HBase (distributed key-value stores), Cassandra (distributed key-value stores), Flink (distributed data stream processing) and Zookeeper (distributed centralized service).

TABLE 3
Six representative systems under testing.

System	Specification (#LOC)	Workload
Hadoop2/Yarn	48	wordcount + kill
HDFS	18	putfile + reboot
HBase	25	write + node crash
Cassandra	17	write
Flink	20	wordcount + cancel
Zookeeper	18	write + node crash

Table 3 lists these six systems. Column 2 gives the number of lines (#LOC) required in a specification in order to adapt each system to a new one. Our communication analysis requires users to manually specify communication patterns for RPC and event dispatch messages (Section 3.1). On average, however, this entails only a 25-line specification for each system. As shown in Figure 5, users are only required to provide the RPC interfaces and methods for dispatching/handling events. CLOUDRAID can then automatically analyze a system under testing to infer the source locations for sending and handling each message. In our evaluation, we use six failure-triggering workloads described in [8] and run the systems using their default configurations (including default logging configurations). These are also common workloads, but errors may be triggered by untimely communication among the nodes.

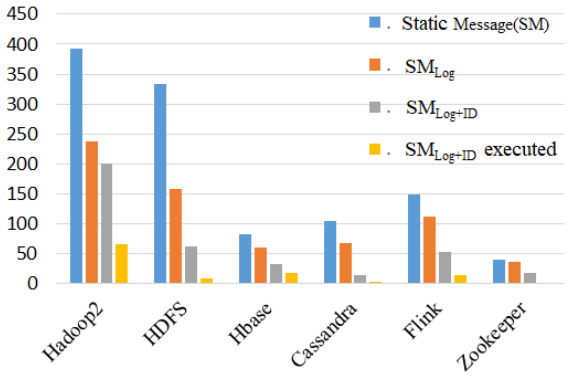

Fig. 10. Number of static messages ("SM"), SM with logs ("SM_{Log}"), SM with logs and IDs ("SM_{Log+ID}"), and SM with logs and IDs that have been executed ("SM_{Log+ID} executed").

Each system runs a 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), but with no noticeable differences 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.

Our evaluation addresses five research questions:

- **RQ1.** How accurate can CLOUDRAID extract message sequences from runtime logs?
- **RQ2.** How much can CLOUDRAID improve the overall testing efficiency?
- **RQ3.** How effective is CLOUDRAID in detecting bugs, especially new bugs?
- **RQ4.** Is our log enhancing technique beneficial?
- **RQ5.** How does CLOUDRAID compare with random message reordering in their bug-finding abilities?

4.1 RQ1: Accuracy

Figure 10 summarizes all the static messages identified in the six systems. There are 393 static messages in Hadoop2/Yarn and 82 static messages in HBase. Our communication analysis can successfully analyze the logging expressions for more than 60% of static messages in all the six systems, except for HDFS (46.8%). We have manually inspected all the static messages that have no logging expressions. Their logs are often optimized away for performance reasons. HDFS frequently reads file systems without logging. Therefore, a large percentage of static messages in HDFS do not have any logging expression.

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. Thus, ID values can effectively distinguish the log instances in different runs. However,

TABLE 4
Statistics of runtime logs. "#Log Instances" is the 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
Zookeeper	3468	20	3524	19	2
Flink	1079	40	3923	54	15

only 39.2% of the static messages in HDFS and 22.1% of the static messages in Cassandra have ID values. HDFS invokes frequently RPC to set/get the state of its master node without ID values. Cassandra mainly prints logs at system startup. For live systems, we process logs for user requests. We analyze further the message logs without ID values: 79.4% print variables such as `size`, 17.2% are daemon processes printing messages such as service start/stop, and the remaining 3.4% are due to bad log quality.

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 Columns 2 and 4. In all the cases except for Cassandra and Flink, the majority of runtime log instances have recorded message events (72.8% for Hadoop2/Yarn, 96.3% for HDFS, 91.0% for HBase and 98.4% for Zookeeper). By comparing Columns 2 and 3, we find that most message log instances have also recorded ID values (74.1% for Hadoop2, 99.6% for HDFS and 60% for HBase). Therefore, these systems provide valuable information for CLOUDRAID to recover accurately the runtime message sequences. Note that Cassandra, Zookeeper and Flink have logged fewer messages and rarely printed ID values in logs. This is because that CLOUDRAID reasons about request logs while these three systems print most of their logs during their system startup processes. Thus, only 1.4% of the runtime log instances have recorded message events with ID values (Figure 10).

The runtime log instances cover about 27.6% of static messages (Column 6 in Table 2 and Figure 10). The other uncovered messages may need to be exercised under a distinct workload (e.g., by executing an "alter table" command for HBase) or a different configuration (e.g., by executing a distinct resource scheduling model in Yarn) or are simply difficult to reach as they are in error handling modules.

Discussion. The accuracy of message sequences extracted by CLOUDRAID varies across the six systems tested. Overall, Hadoop2/Yarn provides the most accurate information in its logs. CLOUDRAID can analyze and process 67.4% of its runtime logs, making it possible to exercise more static messages in Hadoop2/Yarn than the other five systems (Column 6 in Table 4). On the other hand, Cassandra, Zookeeper and Flink rarely log ID values. As a result, CLOUDRAID can only analyze about 1.4% of their runtime log instances, and consequently, cannot accurately recover their runtime message sequences from these logs.

4.2 RQ2: Efficiency

Table 5 reports the times in testing the latest versions of six systems under CLOUDRAID. Column 2 gives the time spent on profiling each system, i.e., running each workload 20 times. In practice, we can obtain logs from live systems without profiling. Column 3 gives the total analysis time, including the times on analyzing the source code and parsing all runtime logs from the 20 runs. Column 4 gives the testing time for triggering all selected orderings. Here, we consider message orderings with two messages only and will examine more messages in Section 4.3.3.

TABLE 5
Analysis and testing times of CLOUDRAID.

System	Profiling (secs)	Analysis (secs)	Trigger (secs)
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
Zookeeper	466.0	60.6	50.8
Flink	721.3	124.209	1813.0

TABLE 6
Message orderings pruned by each analysis. Total is the number of messages orderings. HB is the percentage pruned by HB analysis. Order is the percentage already exercised. ID is the percentage where messages do not log related ID values.

System	Total	% 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%	75.0%
Zookeeper	4	0%	75.0%	0.0%	75.0%
Flink	225	0.1%	12.1%	65.3%	77.5%

CLOUDRAID is very efficient. It finishes its analyses in about 2 minutes in each case (Column 3). In the testing phase (Column 4), Hadoop2/Yarn consumes 6990.2 seconds (1.94 hours). As for Cassandra and Zookeeper, each takes less than 1 minute, since CLOUDRAID can extract only limited information from their runtime logs (Table 4), resulting in only 4 and 2 message orderings to be tested, respectively.

Table 6 shows how CLOUDRAID achieves efficiency by pruning message orderings using different analyses. In Column 2, we have already filtered out the static messages that are not logged with ID values. Otherwise, the number of message orderings for Hadoop2/Yarn would reach 154,449. Overall, CLOUDRAID has successfully pruned 93.6% of the message orderings for Hadoop2/Yarn and 94.4% for HBase (Column 6). Note that HB analysis has pruned only few message orders, as it is difficult to analyze precisely the happen-before order statically in these complex systems. CLOUDRAID prunes efficiently most message orderings by skipping those that have already been exercised (Column 4) and those between unrelated messages (Column 5). We have randomly tested 50 pruned message orderings but failed to 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 23% of them are triggered. For the message orderings not exercised, 82.1% of them do have happen-before relations 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 [31] can further improve efficiency. The remaining 17.9 % are attributed to the fact that our current workloads cannot expose their message orderings.

Discussion CLOUDRAID improves drastically its efficiency by pruning away message orderings that are unlikely to expose errors. However, CLOUDRAID trades soundness for efficiency as we cannot guarantee that all the pruned message ordering will never trigger any error.

4.3 RQ3: Effectiveness

We now show that CLOUDRAID is effective in detecting bugs, especially new bugs, in complex systems.

4.3.1 Finding Existing Bugs

We evaluate CLOUDRAID using the TaxDC Benchmark suite [8] (with one bug per benchmark). The 20 benchmarks in Table 7 are selected because we can manually trigger a failure by changing the order of a message pair in each of these benchmarks. We skip a benchmark in TaxDC if we cannot reproduce the bug in the benchmark manually or if the bug involves timely hardware failure.

TABLE 7
Bugs detected in TaxDC [8].

Category	Bugs
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 MR-5009 HBase-6537 HBase-8940
Not Detected	MR-3006 MR-4099 MR-5009 MR-3721 MR-4842 HBase-10257

CLOUDRAID detects 15 out of the 20 bugs. There are five bugs missed: MR-3006, MR-4099, MR-3721, MR-4842, and HBase-10257. MR-3006 and MR-4099 can only be triggered when instrumenting delays in the middle of their message handlers, which cannot be detected by CLOUDRAID. As for MR-3721 and MR-4842, a key thread message event is not logged. Our log enhancer cannot infer the local ID variables in the corresponding handler and fails to introduce extra logs for the message. Finally, HBase-12507 involves two messages in two subsystems, one in HBase and one in the underlying system Zookeeper. Currently, CLOUDRAID detects bugs caused by messages in one system only.

4.3.2 Detecting New Bugs

We evaluate the ability of CLOUDRAID in detecting new bugs using the six systems given in Table 3. For each system, we select 10 different versions (including the latest, the oldest and eight randomly selected versions). For each system, we apply CLOUDRAID to each version. The same bug appearing in different versions is reported as one bug.

CLOUDRAID has successfully found 31 bugs, comprising 22 already tracked ones and 9 new ones. We have reached this conclusion by using the exceptions raised to search in the bug repositories of these systems. Most of the bugs detected by CLOUDRAID are message order violations (27 out of 31), as expected. CLOUDRAID also detects 4 atomicity violations, which are exposed by concurrent executions of some message handlers caused by message reorderings.

TABLE 8
Bugs detected in the six systems listed in Table 3.

System	#Bugs: new/all		
	Order Violation	Atomicity Violation	Total
Hadoop2/Yarn	6/19	1/2	7/21
HDFS	1/3	0/0	0/3
HBase	1/2	0/2	1/4
Cassandra	0/0	0/0	0/0
Zookeeper	0/0	0/0	0/0
Flink	0/2	0/0	0/0
Total	8/27	1/4	9/31

CLOUDRAID detects the largest number of bugs, i.e., 19 out of 31 in Hadoop2/Yarn but none in Cassandra and Zookeeper (Table 8). Cassandra and Zookeeper are the two systems with the least amount of log information available (Figure 2 and Table 2). The limited log information has largely restricted CLOUDRAID's ability in detecting bugs.

TABLE 9
New bugs detected in the six systems in Table 3. All the bugs have been confirmed by the original developers, with three already fixed.

Bug ID	Bug Type	Status	Patched?	Symptom
YARN-6948	Order	Fixed	✓	Attempt fail
YARN-6949	Order	Unresolved	✗	Wrong state
YARN-7176	Atomicity	Unresolved	✓	Cluster down
YARN-7563	Order	Unresolved	✓	Resource leak
YARN-7663	Order	Fixed	✓	Job fail
YARN-7726	Order	Unresolved	✓	Wrong state
YARN-7786	Order	Fixed	✓	Null Pointer
HBase-19004	Order	Unresolved	✗	Data loss
HDFS-14428	Order	Unresolved	✗	Shutdown abort

Table 9 lists the 9 new bugs detected by CLOUDRAID, with one of these (HDFS-14428) detected thanks to our log enhancing technique. These new bugs may lead to serious failures 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 for six bugs, with three of them already accepted by their original developers.

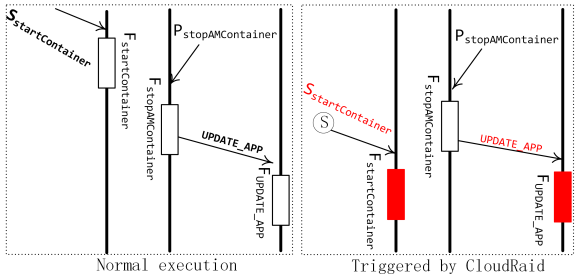

Fig. 11. A new atomicity violation detected by CLOUDRAID. The messages and their handlers triggering the bug are highlighted in red.

Let us examine two new bugs found below.

• **YARN-7176.** CLOUDRAID detects a new atomicity violation in Hadoop2/Yarn, as illustrated in Figure 11. This has been successfully exposed by flipping the normal execution order $S_{\texttt{startAMContainer}} \nrightarrow P_{\texttt{stopAMContainer}}$ so that $P_{\texttt{stopAMContainer}}$, which sends an UPDATE_APP message, is handled first. Thus, its message handler $F_{\texttt{UPDATE\_APP}}$ executes concurrently with the handler $F_{\texttt{startContainer}}$. As a race con-

dition is triggered, an `ArrayIndexOutBoundsException` is thrown, crashing the Yarn daemon process.

• **YARN-7663.** In YARN-7663, CLOUDRAID initially triggered an `InvalidStateTransitionException` error after reordering the `START` message with the `KILL` message. Hence, in our initial patch, we have simply ignored the `START` message if it arrives after the `KILL` message. One original developer accepted our fix by responding with "*Ignoring the START event seems to be appropriate here*". However, he also 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 also `InvalidStateTransitionException` errors in the state machine implementation of YARN. Note that YARN-7703, which is similar to YARN-7726, is not also listed in Table 9. Although the YARN developers have tested the state machine implementation with a large set of unit tests, numerous subtle cases remain uncovered. By iterating over four different versions of our patch (2 months after we reported the bug), the patch was finally accepted and submitted 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), CLOUDRAID can be rather effective. However, for systems that provide limited logs (Cassandra and Zookeeper), CLOUDRAID's bug-finding ability can be restricted.

4.3.3 Reordering Three Messages

TABLE 10
No new bugs found by reordering three messages.

System	#Tests	Time (hrs)	#New Bugs
Haddop2/Yarn	2161	14.6	0
HDFS	93	1.2	0
HBase	314	6.3	0
Cassandra	16	0.1	0
Zookeeper	0	0	0
Flink	323	3.2	0

Table 10 gives the results of reordering three messages. By going from two to three messages, CLOUDRAID has experienced a significant increase in its testing times for all the six systems (e.g., from 2 hours to 14.6 hours for Hadoop2/Yarn) but without any benefit in finding new bugs (Tables 5 and 10). This finding is consistent with a previous study [8], reporting that few distributed concurrency bugs are caused by three or more messages. In general, most concurrent events in distributed systems do not access shared resources. For example, a MapReduce job is divided into thousands of concurrent mapping tasks, where each task communicates only with its own job. Thus, few bugs can be triggered by considering the orderings for three or more messages.

4.4 RQ4: Is CLOUDRAID's Log Enhancer Beneficial?

Figure 12 shows the number of log statements added by our log enhancer. Our log enhancer fails to introduce new logs in Cassandra, Flink and Zookeeper since they rarely print ID

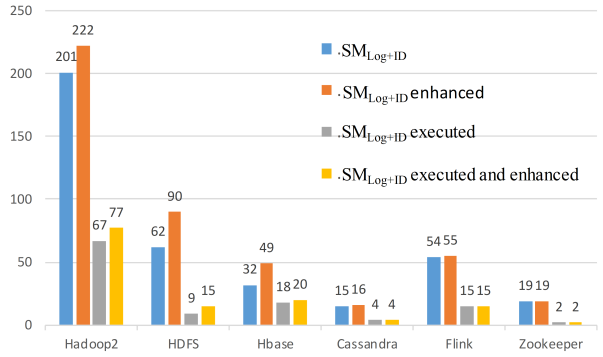

Fig. 12. Number of static messages increased by CLOUDRAID's log enhancer. "SM_{Log+ID}" and "SM_{Log+ID} executed" are from Figure 10 and "SM_{Log+ID} enhanced" and "SM_{Log+ID} executed and enhanced" are their respective versions improved by CLOUDRAID's log enhancer.

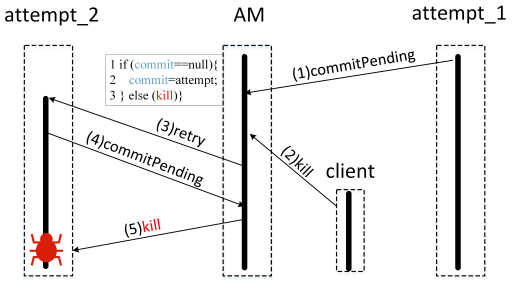

Fig. 13. MR-5009 detected by CLOUDRAID with log enhancement.

variables in their initial logs. Our log enhancer cannot then effectively identify local ID variables and log points for these systems. Our log enhancer has introduced a small number of new logs for the other three systems, Hadoop2, HDFS and HBase. Thus, the runtime overheads due to introduced logging statements are negligible.

The new logs introduced help CLOUDRAID detect more bugs. The four bugs, MR-5009, HBase-6537, HBase-8940 and HDFS-14428 are detected by CLOUDRAID due to log enhancement. Figure 13 illustrates MR-5009, whose code snippet is given in Figure 14. Task `attempt_1` is committing the result and the field `commit` is set accordingly. The client sends a `kill` command to AM to abort `attempt_1`, which is handled by the message handler `AttemptKilledTransition` (Figure 14). Next, the AM node will fork another task `attempt_2` to recommit the result. However, the field `commit` is not reset to `Null` after having aborted the previous attempt. The null test always fails and `attempt_2` is killed, leading to job hang.

This bug can be triggered if the `kill` command is handled after the `commitPending` message. Initially, the message `kill` is not logged. The log enhancer introduces a logging statement in the message handler `AttemptKilledTransition` (Figure 14). This allows CLOUDRAID to discover and exercise the message order `kill` ↣ `commitPending`. The bug is then triggered. This bug can be fixed by resetting the field in the message handler, as shown in lines 7-10 Figure 14.

Discussion. For some systems, our log enhancer can effectively improve log quality and help detect more bugs, with negligible runtime overheads. However, for some systems such as Cassandra, Flink and Zookeeper, the log

```
1. private static class AttemptKilledTransition{
2.   public void transition(TaskImpl task,TaskEvent event){
3.     TaskAttemptId taskAttemptId=event.getTaskAttemptID();
4.     task.finishedAttempts.add(taskAttemptId);
5.     task.inProgressAttempts.remove(taskAttemptId);
6.     //other complex operation
7.     + if ((task.commitAttempt != null) &&
8.     +     (task.commitAttempt == taskAttemptId)) {
9.     +    task.commitAttempt = null;
10.    + }
11.   }
12. }
```

Fig. 14. The code snippet and patch for MR-5009 in Figure 13.

TABLE 11
Bugs found by random message reordering.

System	Time (secs)	#Known Bugs	#New Bugs
YARN	14679	2	0
HDFS	1647.1	0	0
HBASE	3009.6	0	0
Cassandra	119	0	0
Zookeeper	96.5	0	0
Flink	3988.6	0	0

enhancer has little benefits. We can further improve log quality by specifying more ID variables with base types manually.

4.5 RQ5: CLOUDRAID vs. Random Reordering

We compare CLOUDRAID with the strategy of randomly reordering messages. For each system (of its latest version), we choose to execute two messages in a random order. Table 11 shows the results, with random ordering being subject to the same number of runs as CLOUDRAID before.

CLOUDRAID is substantially more effective. With random reordering, we can find only two known bugs, YARN-7786 and YARN-7563, which are also found by CLOUDRAID, but no new bugs at all. CLOUDRAID is also significantly faster. Random reordering can trigger one bug per 3.3 hours, with an average of 373 runs. In contrast, CLOUDRAID can find one bug per 0.34 hours with an average of 41.4 runs.

5 RELATED WORK

We review prior work in detecting distributed concurrency bugs, log analysis, and log enhancement.

5.1 Distributed concurrency bug detection

There is a large body of research on distributed system model checkers [9], [10], [11], [12], [32]. These checkers intercept messages in a system at runtime and permute their orderings exhaustively. While powerful, they suffer from the state-space explosion problem. Recent tools [9], [12] alleviate this problem by adopting state reduction techniques, but may still be unscalable for large state spaces [9].

Liu et al. [33] have recently extended race detection techniques for multi-threaded programs [34], [35], [36], [37], [38], [39] to detect race conditions in distributed systems. Their approach instruments memory accesses and communication events in a system to collect runtime traces at run time. An offline analysis is performed to analyze the happen-before relation among the emory accesses, by using a 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. In [40], its approach mines logs to recover runtime traces without instrumentation, by restricting itself to message orderings involving only two messages. In this paper, we have improved the effectiveness of this earlier approach with two significant extensions. First, we introduce a new log enhancement technique, which allows us to detect bugs that would otherwise be missed. Second, we are now capable of detecting bugs that manifest themselves in message orderings involving an arbitrary number of messages. With these two extensions, we have provided experimental evidence that our framework can find more bugs in new applications.

Fault injection techniques [41], [42], [43], [44], [45], [46], [47], [48], [49] are commonly used to test the resilience of distributed systems. However, they focus on how to inject faults at different system states to expose bugs in the fault handlers. CLOUDRAID can be applied together to detect fault-related concurrency bugs more effectively.

5.2 Log Analysis

Many research efforts [25], [45], [50], [51], [52], [53], [54], [55], [56], [57], [58] mine logs to extract various information, including temporal invariants [51], [53], user request flow [50], [52], system architecture [25], and timing information [56], from distributed systems. The mined information can then be applied to help with better understanding, monitoring, and analyzing complicated distributed systems.

Xu et al. [17] mine console logs from a system and apply machine learning techniques to detect anomaly executions. Mined information such as logged values and logging frequencies is visualized to help users diagnose anomaly behaviors. DISTALYZER [59] compares logs from abnormal and normal executions to infer the strongest association between system components and performance. Iprof [18] extracts request IDs and timing information from logs to profile request latency. Stitch [60] organizes log instances into tasks and sub-tasks, by analyzing relations among the logged ID variables to profile different components in the entire distributed software stack. In contrast, CLOUDRAID mines logs to uncover insufficiently exercised message orderings to detect concurrency bugs effectively.

CRASHTUNER [61] applies a similar log analysis to infer some system meta-info, e.g., the running nodes and tasks/resources associated to each node. This tool makes use of the meta-info to detect crash-recovery bugs, which are triggered by crashing a node where its associated meta-info is being accessed. In contrast, CLOUDRAID applies log analysis to uncover the orderings between communication events for the purposes of detecting distributed concurrency bugs.

5.3 Log Enhancement

Several log enhancing techniques [26], [27], [28], [29], [62], [63] exist to help developers locate the root causes in online systems more effectively (than otherwise). For example, LogEnhancer [26] uses dependence analysis to find variables impacting certain conditional branches and adds these variables in the existing logs. LogAdvisor [27] analyzes

unlogged exceptions in the source code and introduces extra logging statements in the exception handlers.

Unlike these earlier log enhancing techniques, our log enhancer aims to uncover effectively more critical messages that may lead to concurrency bugs. Therefore, our log enhancer introduces log statements for unlogged messages where there exist local ID variables.

6 Conclusion

We present CloudRaid, a simple yet effective tool for detecting distributed concurrency bugs. CloudRaid achieves its efficiency and effectiveness by analyzing message orderings that are likely to expose errors from existing logs. Our evaluation shows that CloudRaid is simple to deploy and effective in detecting bugs. In particular, CloudRaid can test 60 versions of six representative systems in 35 hours, finding successfully 31 bugs, including 9 new bugs that have never been reported before.

Acknowledgments

This work is supported by National Key R&D Program of China (No. 2016YFB1000201), the National Natural Science Foundation of China (61802368, 61521092, 61432016, 61432018, 61332009, 61702485, 61872043), the CCF-Tencent Open Research Fund, and Australian Research Council Grants (DP170103956 and DP180104069).

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[2] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[3] L. George, *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

[4] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[5] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou, "Failure recovery: When the cure is worse than the disease," in *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2490483.2490491

[6] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 249–265. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685068

[7] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 7:1–7:14. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670986

[8] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "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*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 517–530. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872374

[9] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems." in *OSDI*, 2014, pp. 399–414.

[10] H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "Modist: Transparent model checking of unmodified distributed systems," in *6th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2009.

[11] J. Simsa, R. E. Bryant, and G. Gibson, "dbug: systematic evaluation of distributed systems." USENIX, 2010.

[12] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practical software model checking via dynamic interface reduction," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 265–278.

[13] H. Team. (2018) Hdfs architecture guide. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9, 2010.

[15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015. [Online]. Available: http://sites.computer.org/debull/A15dec/p28.pdf

[16] (2018) Wala home page. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page/.

[17] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 117–132.

[18] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "lprof: A non-intrusive request flow profiler for distributed systems." in *OSDI*, vol. 14, 2014, pp. 629–644.

[19] L. Li, C. Cifuentes, and N. Keynes, "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*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 343–353. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025160

[20] ——, "Precise and scalable context-sensitive pointer analysis via value flow graph," in *Proceedings of the 2013 International Symposium on Memory Management*, ser. ISMM '13. New York, NY, USA: ACM, 2013, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/2464157.2466483

[21] T. Tan, Y. Li, and J. Xue, "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*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 278–291. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062360

[22] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 460–473. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950296

[23] (2018) Google protocol buffer. [Online]. Available: https://developers.google.com/protocol-buffers/.

[24] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[25] J.-G. Lou, Q. Fu, Y. Wang, and J. Li, "Mining dependency in distributed systems through unstructured logs analysis," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 91–96, 2010.

[26] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–28, 2012.

[27] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 415–425.

[28] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: enhancing failure diagnosis with proactive logging," in *Presented as part of the 10th {USENIX}*

Symposium on Operating Systems Design and Implementation ({OSDI} 12), 2012, pp. 293–306.

[29] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 565–581.

[30] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "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*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 78:1–78:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389102

[31] Q. Zhou, L. Li, L. Wang, J. Xue, and X. Feng, "May-happen-in-parallel analysis with static vector clocks," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, 2018, pp. 228–240. [Online]. Available: http://doi.acm.org/10.1145/3168813

[32] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code." NSDI, 2007.

[33] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "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, 2017, pp. 677–691.

[34] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 121–133.

[35] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race detection for event-driven mobile applications," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 326–336, 2014.

[36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[37] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: telling the difference with portend," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 185–198, 2012.

[38] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 37–48.

[39] B. Lucia, L. Ceze, and K. Strauss, "Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 222–233, 2010.

[40] J. Lu, F. Li, L. Li, and X. Feng, "Cloudraid: hunting concurrency bugs in the cloud via log-mining," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 3–14.

[41] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "Fate and destini: A framework for cloud recovery testing," in *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011, p. 239.

[42] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva, "On fault resilience of openstack," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 2.

[43] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N. Papakonstantinou, "Setsudō: perturbation-based testing framework for scalable distributed systems," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM, 2013, p. 7.

[44] F. Dinu and T. Ng, "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, 2012, pp. 187–198.

[45] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Visual, log-based causal tracing for performance debugging of mapreduce systems," in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*. IEEE, 2010, pp. 795–806.

[46] H. Team. (2018) Fault injection framework. [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/FaultInjectFramework.html

[47] P. Joshi, H. S. Gunawi, and K. Sen, "Prefail: A programmable tool for multiple-failure injection," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 171–188.

[48] H. S. Gunawi, T. Do, P. Joshi, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and K. Sen, "Towards automatically checking thousands of failures with micro-specifications." in *HotDep*, 2010.

[49] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "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*, ser. ESEC/FSE 2018, 2018.

[50] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," in *ACM SIGPLAN Notices*, vol. 51, no. 4. ACM, 2016, pp. 489–502.

[51] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "Mining temporal invariants from partially ordered logs," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 3, pp. 39–46, 2012.

[52] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2010, pp. 613–622.

[53] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "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, 2011, pp. 267–277.

[54] K. Kc and X. Gu, "Elt: Efficient log-based troubleshooting system for cloud computing infrastructures," in *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE, 2011, pp. 11–20.

[55] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Salsa: Analyzing logs as state machines." *WASL*, vol. 8, pp. 6–6, 2008.

[56] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services." in *OSDI*, 2014, pp. 217–231.

[57] M. Du, F. Li, G. Zheng, and V. Srikumar, "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, 2017, pp. 1285–1298.

[58] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 149–158.

[59] K. Nagaraj, C. Killian, and J. Neville, "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, 2012, pp. 26–26.

[60] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle." in *OSDI*, 2016, pp. 603–618.

[61] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, and L. You, "Crashtuner: detecting crash-recovery bugs in cloud systems via meta-info analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 114–130.

[62] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, 2017.

[63] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 2015, pp. 139–150.

Jie Lu Jie Lu received his BS degree in Computer Science from Sichuan University, China, in 2014. He is currently working toward the PhD degree at the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include program analysis, log analysis and distributed systems.

Feng Li Feng Li received the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2013. She is now working as an associate professor in the Institute of Information Technology, Chinese Academy of Sciences. Her research interests include program analysis and software vulnerability analysis.

Chen Liu Chen Liu received his BS degree in electronic engineering from Tsinghua University, China, in 2017. He is currently working toward the PhD degree at the Institute of Computing Technology, Chinese Academy of Sciences. Chen Liu's research interests include program analysis, deep learning systems and distributed systems.

Lian Li Lian Li received his BSc degree in Engineering physics from Tsinghua University in 1998 and his PhD degree from University of New South Wales in 2007. He is currently a professor in the Institute of Computing Technology, Chinese Academy of Sciences, where he leads the Program Analysis Group.

Lian Li's main research interest focuses on program analysis, more specifically, on program analysis techniques and practical tools for improving software safety and security.

Xiaobing Feng Xiaobing Feng joined the Institute of Computing Technology, Chinese Academy of Sciences since July, 1999. He was working there as an assistant professor, associate professor and then the director of Lab of Advance Compiling Technology. He is now a professor, doctoral advisor and the vice director of Key Laboratory of Computer System and Architecture, ICT. He held the project of developing early versions of binary translation from "X86/Linux" to "LOONGSON/Linux". He also held the project of developing and implementing of compiler and related tools for micro-architectures, such as LOONGSON 2E and LOONGSON 3A.

Jingling Xue Jingling Xue received his BSc and MSc degrees in Computer Science and Engineering from Tsinghua University in 1984 and 1987, respectively, and his PhD degree in Computer Science and Engineering from Edinburgh University in 1992. He is currently a Scientia Professor in the School of Computer Science and Engineering, UNSW Sydney, Australia, where he heads the Programming Languages and Compilers Group.

Jingling Xue's main research interest has been programming languages and compilers for about 30 years. He is currently supervising a group of postdocs and PhD students on a number of topics including programming and compiler techniques for multi-core processors and embedded systems, concurrent programming models, static and dynamic program analysis for detecting bugs and security vulnerabilities.