ReproLite : A Lightweight Tool to Quickly Reproduce Hard System Bugs

Kaituo Li
University of Massachusetts, Amherst
kaituo@cs.umass.edu

Pallavi Joshi
NEC Laboratories America
pallavi@nec-labs.com

Aarti Gupta
NEC Laboratories America
agupta@nec-labs.com

Malay K. Ganai
NEC Laboratories America
tomalgan@gmail.com

Abstract

Cloud systems have become ubiquitous today – they are used to store and process the tremendous amounts of data being generated by Internet users. These systems run on hundreds of commodity machines, and have a huge amount of non-determinism (thousands of threads and hundreds of processes) in their execution. Therefore, bugs that occur in cloud systems are hard to understand, reproduce, and fix. The state-of-the-art of debugging in the industry is to log messages during execution, and refer to those messages later in case of errors. In ReproLite, we augment the already widespread process of debugging using logs by enabling testers to quickly and easily specify the conjectures that they form regarding the cause of an error (or bug) from execution logs, and to also automatically validate those conjectures.

ReproLite includes a Domain Specific Language (DSL) that allows testers to specify all aspects of a potential scenario (e.g., specific workloads, execution operations and their orders, environment non-determinism) that causes a given bug. Given such a scenario, ReproLite can enforce the conditions in the scenario during system execution. Potential buggy scenarios can also be automatically generated from a sequence of log messages that a tester believes indicates the cause of the bug. We have experimented ReproLite with 11 bugs from two popular cloud systems, Cassandra and HBase. We were able to reproduce all of the bugs using ReproLite. We report on our experience with using ReproLite on those bugs.

SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA.
ACM 978-1-4503-3252-1.
http://dx.doi.org/10.1145/2670979.2671004

Categories and Subject Descriptors D.2.5 [*SOFTWARE ENGINEERING*]: Testing and Debugging; C.2.4 [*COMPUTER-COMMUNICATION NETWORKS*]: Distributed Systems

Keywords Debugging, Lightweight, Cloud Computing, Hard System Bug

1. Introduction

Large-scale distributed systems (or cloud systems) that run on hundreds of machines (nodes) are increasingly being used by today's software companies to host, process, and analyze the tremendous amount of data being generated by their enormous numbers of users (tens of thousands or even millions of users). These systems are complex with huge amounts of non-determinism (thousands of threads and hundreds of processes), not only within each process but also across different processes. Thus, it is hard to find, reproduce, and understand bugs in distributed systems [18]. Even after a bug has been discovered in an execution, a subsequent execution of the same system under the same environment and configuration might not result in the bug. Bugs often occur under specific conditions - specific workloads, specific schedules of operations (events) within a node or across different nodes, specific external non-deterministic events (e.g., garbage collection, node crash, network failures, etc.) in the execution environment or in the hardware used by the system. Finding the conditions responsible for a bug and enforcing those conditions during system execution to reproduce the bug are hard.

The state-of-the-art of debugging cloud systems in the industry is logging messages during execution, and referring to the messages later when there is an error. The log messages can often provide hints regarding the cause of the error. But, even after forming a hunch regarding the cause of the error, it takes significant effort to validate the hunch (e.g., inserting sleeps to try to enforce a particular order of events, physically bringing network links down, etc.). Many a time testers fix their code based on their hunch, and if the error or the bug does not show up during execution after the fix, they declare victory over the bug. Just because the bug did not show up after the fix does not mean that it has really been fixed – the executions subsequent to the fix might not have exhibited the conditions necessary for the bug (which might still be feasible after the fix).

There has been a lot of work [16, 21, 24, 25, 27] on record-and-replay systems that record the partial-order among non-deterministic operations in a system execution (e.g., order of racing memory accesses, network messages, preemptions, OS signals, etc.) and later replay the partial order to deterministically reproduce the observed execution. However, logging involves a significant overhead, and developers do not want to incur a huge overhead by aggressively recording all non-determinism in an execution. Instead of increasing logging for a system, we help testers to form hypotheses regarding the cause of a bug from incomplete logs (that do not record all non-determinism), and validate those hypotheses quickly with little execution overhead.

ReproLite augments the already widespread process of using logs for debugging by providing support for validating the cause of a bug that testers infer from execution logs. It has a Domain Specific Language (DSL) in which testers can easily and quickly express the potential cause of the bug (workload, schedule of system events, external events). ReproLite can then reproduce the potential buggy scenario expressed in the DSL. It instruments the system binaries so that it can observe the system events in the DSL during execution. As the system executes, ReproLite executes the workload specified by the tester, and enforces the specified order among the system events and external events. If the bug is reproduced by the specified scenario, then the tester has validated her hunch and the specification for the bug allows her to communicate her observations to the developers in a succinct and precise manner. ReproLite also gives the developers a means to reproduce the bug over and again that can help them to understand and fix the bug. Even if the hunch was wrong and the bug did not get reproduced, the tester can learn from the execution enforced by ReproLite, update her conjecture regarding the cause of the bug, and re-validate the updated conjecture. Note that our DSL allows ordering of events from different processes and workloads, and not only for events in a single process as in previous work (e.g., Concurrit [14]). Section 6 has more details and comparison against related work.

Given a sequence of log messages that the tester believes reflects the order of system events that caused the bug, ReproLite can automatically generate the corresponding scenario in the DSL and constrain the execution in a way that the given log messages execute in the given order. In our experiments with real-world cloud system bugs, we found that for most (70 %) of the bugs that we studied, part of the cause of the bug was hidden in the order of a few log messages. Thus, if the tester can find the relevant log messages, then ReproLite can reproduce the bug using the order of those messages.

The main contributions of our work are:

1. We have designed and implemented a DSL in which testers can quickly express potential buggy scenarios by expressing the specific workload needed to trigger the bug, system events and their order that would lead to the bug, and any external events like garbage collection, node crash, or network failures that are also responsible for the bug. The DSL makes it easy for testers to specify all aspects of a bug (workload, event order, etc.) in one place.
2. We have implemented a tool that can reproduce a buggy scenario expressed in the DSL. The tool can execute the specified workload, intercept the system events expressed in the DSL and enforce them to execute in the given order, and also trigger or emulate specified external events during execution. The DSL and the reproduction engine augment the widespread debugging process of using logs by enabling testers to validate the conjectures that they form from execution logs.
3. We experimented with using ReproLite to debug 11 bugs from two real-world cloud systems: Cassandra and HBase. We could reproduce all of the bugs using ReproLite. We report on our experience in Section 4. An interesting finding was that for most (70 %) of the bugs, the cause of the bug was hidden in the log messages. After we found the right set of log messages (as is done in the current practice of debugging using logs), ReproLite was able to automatically generate system event definitions and their order in the buggy scenario, replay the scenario, and exhibit the bug.

In the rest of the paper, we provide an overview of ReproLite using a real-world cloud system in Section 2, explain the internals of ReproLite in Section 3, and evaluate ReproLite on bugs in real-world cloud systems in Section 4. We compare our work with other related work in Section 6.

2. Overview

In this section, we show how ReproLite works using a bug from HBase [7], a distributed data store. HBase hosts data as tables which are in turn divided into regions. Each region is served by a particular kind of HBase node known as a regionserver. All read and write requests to a region are handled by the regionserver hosting that region. When a region grows and becomes larger than a pre-defined size (that can be configured), it is split into two by the regionserver. There is a master node in HBase that monitors the regionservers. When a regionserver goes down, the master reassigns the regions hosted by the regionserver to other alive regionservers. HBase uses another distributed system called

```
bug.sc:

E1[node]='regionserver'
E1[stack]='createNodeSplitting'

E2[node]='master'
E2[stack]='addSplittingToRIT
           handleRegion'

E3[node]='master'
E3[stack]='NavigableMap.remove
         ServerShutdownHandler.process'

E4[node]='master'
E4[stack]='regionOffline
        AssignmentManager.nodeDeleted'
```

Figure 1. Relevant events in the HBase bug HBASE-6070

```
bug.sc:
       W1='create-table-cf.sh test cf'
       W2='insert.sh 20 test cf'
       W3='flush.sh test'
       W4='compact.sh test'

       S1=block after E1
       S2=unblock after E1

       X1=node-down 'regionserver'

       W1 * W2 *
          (W3 * W4 ||
             E1 * S1 * E2 * S2 * X1
                     * E3 * E4)
```

Figure 2. The buggy scenario for HBASE-6070 in $\mathbb{RT}$. The definition of workload components and events are provided in Figures 5 (Section 3.1) and 1.

ZooKeeper [2] to manage its configuration (e.g., which regions have been assigned to which regionservers). Hadoop Distributed File System [6] is used to store and replicate data.

There is a bug in HBase (HBase-6070 [1]) that occurs when a regionserver is starting to split a region that has grown beyond the pre-defined limit. Just as the regionserver starts to split by changing the configuration information in ZooKeeper regarding the region, the regionserver crashes. There is a callback that runs in the master when the master detects that the regionserver has crashed, and there is another callback that also runs in the master when the master detects that the region that was beginning to split has gone offline. If the two callbacks interleave in a certain manner, then the master mistakenly believes that the region has already split, and does not re-assign the region to another regionserver. As a result, clients lose access to any data in that region. This is a serious bug and has been classified as "Major" by HBase developers.

HBase-6070 occurs under very specific conditions: a regionserver crashes just when it starts splitting a region, and the two callbacks in the master in response to the crash and the offlining of the region interleave in a manner that misleads the master about the state of the region being split. Enforcing these conditions is hard – one would have to modify the source code to kill the node after it starts splitting a region, and add sleep's to try to enforce the specific interleaving between the callbacks (which might not be always be successful). Using the DSL $\mathbb{RT}$ of ReproLite, a tester can quickly express the scenario that leads to HBase-6070 as in Figure 2. '*' and || express scheduling constraints. $p * q$ means execute p then q, and $p||q$ means execute p and q simultaneously (Section 3.1).

Figure 2 indicates that the workload component W1 should be executed followed by W2. The workload components are described in detail later in Section 3.1. W1 creates a table and a column family, and W2 inserts data into the column family. The amount of data inserted is more than the amount that would trigger region splitting (maximum size of regions can be configured by the tester). W3 is then executed that flushes the table to disk, and then W4 that compacts the table. During flushing and compaction, if the region is found to have grown beyond its limit, splitting of the region is initiated.

As W3 and W4 execute, the scenario in Figure 2 enforces certain constraints on the execution in HBase. The event E1 should execute in a regionserver, followed by E2 in the master, and then X1 that crashes the regionserver. E1 and E2 indicate the beginning of a region split. The events have been defined in Figure 1. An event is either the execution of an operation in the given system (e.g., E1) or an operation or phenomenon in the execution environment (e.g., garbage collection and network failures). Section 3.1 explains how events can be defined in the DSL. S1 and S2 in Figure 2 specify further scheduling constraints, ensuring that the regionserver crashes before proceeding any further after E1. The "block after" and "unblock after" constructs in S1 and S2 apply to the thread that has executed E1. S1 leaves the thread blocked after it completes E1. S2 is called to unblock the thread afterwards. The master should then execute E3 followed by E4 for the bug to occur.

Given the scenario in Figure 2, ReproLite instruments HBase binaries so that it can observe when splitting of a region begins, and when the callbacks in the master (that is, events E1, E2, E3, and E4) execute. Since there are only a few events that ReproLite has to track (as is the case with most of the bugs), the overhead of instrumentation is minimal. ReproLite executes the workload (W1, W2, W3, W4) specified in the scenario, tracks and enforces the order between E1, E2, E3, and E4, and also brings down the regionserver (X1) after E2. A tester can use ReproLite to reproduce HBASE-

```
In regionserver's logs:

2014-01-24 15:54:30,642 DEBUG
regionserver.SplitTransaction (SplitTransaction.
java:createNodeSplitting(857)) - regionserver:
60020-0x143c607d6120004 Creating ephemeral node
for 5d755f81b0421a673223facdcb7bfecc in SPLITTING
state

In master's logs:

2014-01-24 15:54:44,256 DEBUG master.
AssignmentManager (AssignmentManager.java:
handleRegion(699)) - Handling transition=
RS_ZK_REGION_SPLITTING, server=d2.nec-labs.com,
60020,1390596846260, region=
5d755f81b0421a673223facdcb7bfecc

2014-01-24 15:55:27,276 DEBUG handler.
ServerShutdownHandler (ServerShutdownHandler.
java:process(275)) - Removed test,,1390596882207.
5d755f81b0421a673223facdcb7bfecc. from list of
regions to assign because in RIT; region state:
SPLITTING

2014-01-24 15:55:26,030 DEBUG master.
AssignmentManager (AssignentManager.java:
nodeDeleted(1122)) - Ephemeral node deleted,
regionserver crashed?, clearing from RIT;
rs=test,,1390596882207.
5d755f81b0421a673223facdcb7bfecc. state=SPLITTING,
ts=1390596884258, server=d2.nec-labs.com,60020,
1390596846260
```

Figure 3. Log messages relevant for HBASE-6070. Components of the log messages relevant to the bug are italicized and are in boldface.

6070 repeatedly in order to understand how to fix it. After the fix, the tester can again use ReproLite to ensure that HBASE-6070 has indeed been fixed. Thus, the $\mathbb{RT}$ scenarios can serve as regression tests.

In order to write the potential scenario for a bug, a tester needs to have a hunch regarding the cause of the bug. The logs of the execution in which the bug unexpectedly occurred are a good place to start. This is in fact how bugs are found and understood in cloud systems. After forming an intial conjecture regarding the cause, the tester can quickly express the cause in the DSL and validate using ReproLite. Even if the bug is not reproduced using the intial conjecture, the tester can refine it and re-validate it based on what she obseved and learnt from the execution enforced by ReproLite for the intial conjecture. In fact, for most (70 %) of the bugs that we experimented with, we saw that the logs often contained the cause of the bug in them. For example, Figure 3 shows the log messages that indicate the system events and their order that caused the bug. The first log message corresponds to event E1 (Figure 1), the second to E2, third to E3, and fourth to E4. Once a tester has figured out a set of log messages that potentially indicate the cause of the bug, ReproLite can automatically create the potential buggy scenario and reproduce it.

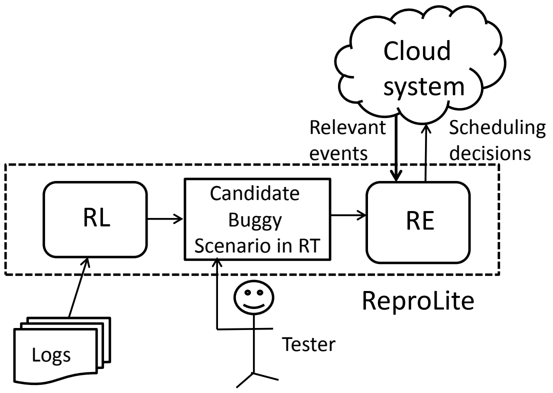

Figure 4. Overall architecture of ReproLite

3. ReproLite Internals

In this section, we explain the architecture (Figure 4) and workings of ReproLite. There are three key components of ReproLite: $\mathbb{RT}$, the domain specific language (DSL) that enables testers to easily and quickly express their various hunches regarding the workload, events, and their execution order that caused a given bug, $\mathbb{RE}$, the reproduction engine that enforces the given execution order, and $\mathbb{RL}$, the log analyzer that can automatically generate potential buggy scenarios in $\mathbb{RT}$ from execution logs that can be inspected and further modified by a tester. The following sections explain the three components in more detail.

3.1 $\mathbb{RT}$: The DSL to Express Potential Buggy Executions

$\mathbb{RT}$ enables a tester to easily and quickly express the scenario (workload, event orders, external events) that could potentially have led to a given bug. The reproduction engine (Section 3.2) enforces the expressed scenarios, and thus, helps the tester to find out the specific scenario and conditions that caused the bug. We will also show later (Section 3.3) how a tester can infer hints regarding the buggy scenario from execution logs.

3.1.1 Expressing workload

In order to express an execution scenario, we need to specify the workload that was executed in that scenario. Sometimes, a bug might occur without any workload (e.g., a bug that occurs when the system nodes are booting up) in which case we need not specify the workload, but often a bug is triggered only under specific workloads (e.g., only when a column family is dropped in Cassandra or only when a table is disabled in HBase). To specify a workload, we need to create (if not already present) and provide the names of executables for different kinds of client requests in the workload. For example, let's say that we want to create a table and a column family and insert some data into that column family in HBase. Then, we can create the executables as shown in Figure 5 to create a table and a column family, and to insert

```
insert.sh:
  N=$1;  tb=$2;  cf=$3
  for ((i = 1; i <= N; i++)); do
    echo put \'$tb\', \'row\', \'$cf:a$i\'
        ,\'v$i\' >> tmp; done
  echo exit >> tmp
  hbase shell tmp; rm tmp
```

Figure 5. Insert data into a table and a column family in HBase. Scripts to create table and column family (create-table-cf.sh), flush (flush.sh) and compact (compact.sh) a table are similarly implemented.

data into a column family. We use the names of the executables to specify our workload as shown under `bug.sc` (part of the buggy scenario written using $\mathbb{RT}$) in the same figure. `W1 * W2 * W3 * W4` indicates the sequence of `W1`, `W2`, `W3`, and `W4`. Since the kinds of requests that can be made against a system (e.g., create table, insert into table, delete table, etc. in Cassandra) are limited, we can create executables for all (or most) of those kinds of requests, and let different testers re-use those executables for their workloads.

3.1.2 Expressing event orders

Apart from specifying the workload in a buggy scenario, $\mathbb{RT}$ also lets a tester specify the events and their order during execution that could have led to the given bug. Since there is a tremendous amount of non-determinism during execution in a cloud system, many a time a bug can occur under a specific schedule of events and might not show up under other schedules of those events. For example, in HBase, we cannot access data in a region (a chunk of a table) at all when the node hosting that region (a regionserver) crashes just when it had started to split the region, and when the two callbacks regarding the crash in the master node (node monitoring all regionservers) interleave in a specific manner (HBASE-6070 [1]). There are multiple events across different nodes that need to execute in a specific order for the HBase bug to show up. A tester can specify such events and their execution order in $\mathbb{RT}$.

To specify an event, we specify its execution context. For example, to specify the starting of the splitting of a region in a regionserver (a relevant event in HBASE-6070 as described before), we can specify part of its execution context – the node in which the event is executing (`regionserver`) and the method that is executing (`SplitTransaction.createNodeSplitting`). Figure 1 illustrates how the events in HBASE-6070 can be expressed in $\mathbb{RT}$. There are four relevant events – `E1`, `E2`, `E3`, and `E4`. For each event, different aspects of its execution context have been specified – the node that the event is executing on and a part of its execution stack. There are other aspects of the context that can be expressed, e.g., the arguments to the executing method, thread ID (not the runtime ID but an integer that distinguishes the thread from other threads), the remote node that sent data or is going to receive data (in case of network I/O), etc.

To specify an event order, a tester can sequence different events using the sequence operator `*`. For example, `E1 * E2 * E3 * E4` indicates the order in which `E1` executes before `E2`, `E2` before `E3`, and `E3` before `E4`. The tester can also use the parallel operator (||) to express that two events should execute concurrently and that the order between them is not important. For example, `E1 * (E2 || E3)` indicates the order in which `E1` executes, and then `E2` and `E3` execute in either order. In addition, one can also block the execution of a thread after a specific event, and direct to resume its execution later after another event. For example, in HBASE-6070, after executing `E1`, the thread in `regionserver` should proceed only after `E2` has been executed in `master`. A tester can specify this constraint using `E1 * S1 * E2 * S2 * E3 * E4`, where `S1='block after E1'` and `S2='unblock after E1'`. The `block` and `unblock` keywords provide a convenient mechanism to express certain scheduling constraints.

3.1.3 Expressing external events

Lastly, $\mathbb{RT}$ enables one to also specify external non-deterministic events that can occur in the execution environment (e.g., garbage collection) or in the hardware on which the system is executing (e.g., node crash, network failure between nodes, disk failures, etc.). Sometimes, a bug would occur only when a non-deterministic external event unexpectedly occurs during execution, e.g., when the region-server crashes when starting to split a region in HBASE-6070 as explained earlier. To specify an external event, a tester has to use the appropriate keyword provided in the DSL : `node-down` for a node crash and `node-up` for rebooting a node if it is down, `nw-link-down` for bringing a network link down and `nw-link-up` for bringing a link up if it is down, `gc` for invoking the garbage collector, etc. We can also specify the node that we want to crash/reboot, or the link that we want to bring down/up, e.g.. `node-down regionserver` would crash a regionserver, and `nw-link-down master regionserver` would bring the network between the master and a regionserver down.

To put it all together, Figure 2 shows how the buggy scenario for HBASE-6070 can be written in $\mathbb{RT}$.

3.1.4 Formalization

Figure 6 shows the syntax of $\mathbb{RT}$. Some of the details are omitted for space constraints.

Figure 7 shows the operational semantics of ReproLite. We consider the state of ReproLite to be a tuple consisting of the following:

- σ: the state of the SUT (software under test).

WL	::=	W SP | WL SP W	*Workloads*
W	::=	WID ‘=’ ⟨ STRING ⟩	*Workload script name with args*
WID	::=	‘W’ ⟨ INTEGER ⟩	
SP	::=	‘\n’	
ENL	::=	EN SP | ENL SP EN	*Internal Events*
EN	::=	EID “[node]” ‘=’ ⟨ STRING ⟩	*Node ID of an event*
ESL	::=	ES SP | ESL SP ES	
ES	::=	EID “[stack]” ‘=’ ⟨ STRING ⟩	*Stack trace as string*
EID	::=	‘E’ ⟨ INTEGER ⟩	
···			*Similar rules for thread ID, remote node, etc.*
XL	::=	X SP | XL SP X	*External events*
X	::=	XID ‘=’ XOP ⟨ STRING ⟩	
XOP	::=	“node-down” | “garbage collection” | ···	
XID	::=	‘X’ ⟨ INTEGER ⟩	
SCL	::=	SC SP | SCL SP SC	*Scheduling constraints*
SC	::=	SID ‘=’ SAction SOrder EID	*Block/unblock a thread before/after event EID*
SAction	::=	“block” | “unblock”	
SOrder	::=	“before” | “after”	
SID	::=	‘S’ ⟨ INTEGER ⟩	
BugSc	::=	WL ENL XL SCL PREXPR	*The buggy scenario*
PREXPR	::=	SQEXPR | PREXPR “||” SQEXPR	*Events and workloads to execute in parallel*
SQEXPR	::=	EXPR | SQEXPR ‘*’ EXPR	*Events and workloads to execute in sequence*
EXPR	::=	WID | EID | XID | SID	

Figure 6. Syntax of buggy scenario in $\mathbb{RT}$. Some details are omitted for brevity.

$$\frac{\langle(\sigma,\mu),WID\rangle \xrightarrow{w} (\sigma',\mu')}{\langle(\sigma,\mu,\beta),WID\rangle \rightarrow (\sigma',\mu',\beta)}\ \textit{Execute a workload} \qquad \frac{\exists t(matches(\mu[t],EID) \wedge \langle(\sigma,\mu),\mu[t]\rangle \xrightarrow{e} (\sigma',\mu'))}{\langle(\sigma,\mu,\beta),EID\rangle \rightarrow (\sigma',\mu',\beta)}\ \textit{Execute an internal event}$$

$$\frac{\exists t(matches(\mu[t],EID) \wedge (\beta'{=}\beta+t))}{\langle(\sigma,\mu,\beta),block\ before\ EID\rangle \rightarrow (\sigma,\mu,\beta')}\ \textit{Block before} \qquad \frac{SID = block\ after\ EID \quad \exists t(matches(\mu[t],EID) \wedge \langle(\sigma,\mu),\mu[t]\rangle \xrightarrow{e} (\sigma',\mu') \wedge (\beta'{=}\beta+t))}{\langle(\sigma,\mu,\beta),EID*SID\rangle \rightarrow (\sigma',\mu',\beta')}\ \textit{Block after}$$

$$\frac{\langle(\sigma,\mu,\beta),EX1\rangle \rightarrow (\sigma',\mu',\beta') \quad \langle(\sigma',\mu',\beta'),EX2\rangle \rightarrow (\sigma'',\mu'',\beta'')}{\langle(\sigma,\mu,\beta),EX1||EX2\rangle \rightarrow (\sigma'',\mu'',\beta'')}\ \textit{Execute in parallel}(1)$$

$$\frac{\langle(\sigma,\mu,\beta),EX2\rangle \rightarrow (\sigma',\mu',\beta') \quad \langle(\sigma',\mu',\beta'),EX1\rangle \rightarrow (\sigma'',\mu'',\beta'')}{\langle(\sigma,\mu,\beta),EX1||EX2\rangle \rightarrow (\sigma'',\mu'',\beta'')}\ \textit{Execute in parallel}(2)$$

Figure 7. Operational Semantics of $\mathbb{RT}$. Some rules are omitted for brevity.

- μ: a map from each thread to the next relevant internal event to execute in the thread. Different threads might be running in different processes in different nodes.
- β: a queue of blocked threads. A thread can be blocked and unblocked by scheduling constraints (Figure 6).

We utilize the following definitions in the semantics.

- The transition function $\xrightarrow{w}$ ($\langle(\sigma,\mu),WID\rangle \xrightarrow{w} (\sigma',\mu')$) executes workload *WID* against the SUT, and modifies the SUT state σ and μ appropriately.
- Similarly, the transition function $\xrightarrow{e}$ ($\langle(\sigma,\mu),ev\rangle \xrightarrow{e} (\sigma',\mu')$) executes internal event *ev* in the SUT, and modifies the SUT state σ and μ appropriately.
- `matches(ev, EID)` returns true if the context of event ev matches that of EID given in the $\mathbb{RT}$ scenario.
- The function $-/+$ removes from/adds a thread to β.

3.2 $\mathbb{RE}$: The Reproduction Engine

Given a potentially buggy scenario in $\mathbb{RT}$, the reproduction engine tries to create that scenario during system execution. It executes the specified workload, takes control over the system execution to enforce the specified schedule of system events, and also triggers or emulates specified external events (e.g., node crash and garbage collection). To execute the given workload, $\mathbb{RE}$ invokes the different executables specified for the workload. Enforcing the given schedule of events and emulating external events are not as straightforward, and are explained in subsequent sections (Sections 3.2.1 and 3.2.2, respectively).

3.2.1 Enforcing a specific event order

To enforce a specific order of events, $\mathbb{RE}$ first needs to know when those events occur during execution. For this, we instrument (add hooks to) the system binary so that the hooks can notify the reproduction engine when relevant events occur. For example, to know when a regionserver starts splitting a node in HBase (event `E1` in Figure 1), we add a hook before the method '`createNodeSplitting`'. The hook notifies $\mathbb{RE}$ when a regionserver is about to execute the method. Similarly, to know when a callback executes in the master in response to a region becoming offline (event `E4`), we add a hook before the method `regionOffline`. The hook notifies when the method is going to be executed in the context of the execution of `AssignmentManager.nodeDeleted`.

We target cloud systems written in Java for this work, and use AspectJ [9] to instrument Java bytecode. Appropriate instrumentation tools can be used for systems written in other languages. For each event specified in a given scenario, we automatically generate an aspect that would notify the reproduction engine when that event occurs. For example, Figure 8 shows the aspect that is generated for event `E4` in Figure 1. The aspect imports necessary classes from the definition of `E4`. `E4` indicates that '`regionOffline`' executes in the context of '`AssignmentManager.nodeDeleted`'. Thus, the generated aspect imports the '`AssignmentManager`' class. The pointcut `p()` captures what the tester had specified should be present in the execution stack of `E4`. The DSL $\mathbb{RT}$ also allows testers to specify the types of arguments for the executing method, and values for any of those arguments. We incorporate the given argument information in the generated aspect using the `args` construct in AspectJ. Since a scenario typically has a few (less than 10 in our experiments (Section 4)) events, we have to instrument only a few execution points. The overhead due to instrumentation is thus low.

The code within the aspect generates other information regarding the intercepted execution point, e.g., node ID, thread ID, remote node ID and ports in case of network IO, etc. During execution, the generated information regarding the intercepted execution point is sent to $\mathbb{RE}$. $\mathbb{RE}$ matches the information against the event definitions in the given buggy scenario to decide if the intercepted execution point corresponds to one of the events in the scenario.

As the system under consideration executes, the aspects notify the reproduction engine when a potentially relevant event executes in a node in the system. The reproduction engine runs as a separate process (likely on a different machine). The system nodes (with aspects in their binaries) communicate with the engine via RPC. A system node sends relevant information for an event to $\mathbb{RE}$ (`RE.order(ctx)` in Figure 8). $\mathbb{RE}$ matches this information against the next event specified in the buggy scenario. If there is a match (e.g., the aspect `p()` in Figure 8 executes in `master` and the sequence of events till `E3` have already been observed for the scenario in Figure 2), then $\mathbb{RE}$ allows the event execution to proceed. Otherwise, if there is a match against an event later in the schedule (e.g., `p()` in Figure 8 executes and events till only `E2` have been observed), then $\mathbb{RE}$ blocks the RPC call from the node effectively blocking the node from executing the event. This is because the given scenario requires the current event to execute only after other not-yet-encountered events have executed. The current event would be allowed to proceed after those other events have been encountered and executed. If there is no match with the next event or any future event, $\mathbb{RE}$ lets the intercepted event proceed as usual.

```
import org.apache.hadoop.hbase.master.
           AssignmentManager;

pointcut p() : execution(*
 *.regionOffline(..)) &&
    cflow(execution
      (* AssignmentManager+.
                nodeDeleted(..)));

before() : p(){
 Context ctx = new Context(
                     thisJoinPoint);
 ... RE.order(ctx); ... }
```

Figure 8. Aspect generated for event `E4` in Figure 1

Sometimes the specified scenario might not be feasible. An event `E` that is expected next in the scenario might not show up during system execution. If $\mathbb{RE}$ does not observe the next event within a pre-defined amount of time (e.g., 20 minutes), it gives up on reproducing the given scenario. But, if $\mathbb{RE}$ cannot reproduce a scenario, then it does not necessarily mean that the scenario is infeasible. It might so happen that a given event definition in the scenario matches multiple system events, but a bug occurs only for some of the system events that match. For example, let's say that event `E3` in Figure 1 is defined as `E3[node]='master'` and `E3[stack]='NavigableMap.remove'` (and not `E3[stack]='NavigableMap.remove ServerShutdownHandler.process'`). Thus, according to the new definition of `E3`, any execution of `NavigableMap.remove` in the master node would match `E3`. If the reproduction engine orders an execution of `NavigableMap.remove` that is not in the context of `ServerShutdownHandler.process` before E4 (Figure 1), then no bug might show up. When 𝔼 does not reproduce the given buggy scenario, the tester can refine the events in the scenario (e.g., add `ServerShutdownHandler.process` to E3's stack) in order to help ℝ𝔼 to better identify the system events involved in the bug.

3.2.2 Emulating non-deterministic external events

The reproduction engine can also trigger or emulate non-deterministic events that can occur in the execution environment (e.g., garbage collection) or in the hardware on which the system executes (e.g., node crash, network failures, disk errors, etc.). To trigger a garbage collection, ℝ𝔼 calls `System.gc()` that advises the Java runtime to perform garbage collection. To emulate a node crash, we kill the process for the node. For network failures, we intercept the network IO calls for communication between the nodes or ports that we want to fail, and instead of letting the calls execute as they normally would, we fail them with IO exceptions. We emulate disk errors similarly. The details of emulating hardware failures have been described in our previous work [17, 19, 20, 22].

3.3 ℝ𝕃: The Log Analyzer

In order to write and validate a scenario that could lead to a bug, a tester needs to have a hunch regarding the conditions that cause the bug. If the tester has no idea as to what leads to the bug, then the tester can use logs from an erroneous execution to get hints regarding the cause of the bug. Cloud system developers often employ logging to record information regarding an execution that they use for debugging in case of errors. This is in fact how debugging is done in most of the cloud systems industry. ReproLite augments this already prevailing debugging approach by enabling developers to quickly prototype and validate hunches that they form from examining logs.

The log analyzer helps testers to process logs and compare logs from different executions in various ways, and also automatically generate scenarios in ℝ𝕋 from a sequence of log messages. A log message can have different components: timestamp, file name, line number, class name, method name, argument string describing partial execution state, remote node, etc. ℝ𝕃 enables a tester to extract a subset of components from log messages. Testers can also stitch together multiple logs according to the timestamps of their log messages in order to obtain a global view of the execution. They can also find diff's of logs from two different executions (e.g., erroneous and correct executions) to understand which of the differences between the two executions could have caused the bug.

After examining and comparing logs using ℝ𝕃, testers can narrow their focus to a subset of log messages that can potentially reveal the cause of the bug (e.g., the log messages in the diff of erroneous and correct executions). Given a sequence of such log messages (possibly from logs in multiple nodes), ℝ𝕃 can automatically generate a scenario in ℝ𝕋 that would enforce an execution in which those log messages would execute in the order in the given sequence. For example, Figure 3 shows the relevant log messages for HBASE-6070 that a tester can find using the filtering and diff'ing techniques described above. ℝ𝕃 can automatically generate event definitions in ℝ𝕋 from the given log messages as shown in Figure 9.

For each relevant log message, ℝ𝕃 parses the message, and extracts different components out of it (e.g., file name, line number, class and method names, argument string). Since we target cloud systems written in Java, we target popular logging packages in Java (e.g., log4j and Java Logging). Using the log message pattern provided in the system configuration, we parse messages to extract their components. For example, following is a log4j pattern in Cassandra that can be used to parse log messages: %5p [%t] %dISO8601 %F:%C:%M (line %L) %m%n. The pattern specifies that the first term in a log message is the log level (e.g., info, debug, error, etc.), the second term is the thread name that logged the message, the third term is the date in ISO8601 format, the fourth term is the file name, the class name, and the method name, and the fifth term is the line number of the statement that generated the message. In the end, there is an arbitrary string that is provided as an argument to the log statement.

ℝ𝕃 generates appropriate event definitions from the parsed relevant log messages, and generates a candidate scenario in ℝ𝕋 in which the events execute in the same order as the respective log messages. For example, consider the first log message in Figure 3. After parsing the log message as described above, we find that the different components of the message are: (i) `SplitTransaction.java` (file name), (ii) 857 (line number), (iii) `SplitTransaction` (class name), (iv) `createNodeSplitting` (method name), and (v) `regionserver ...in SPLITTING state` (log statement argument). We strip off substrings in the log statement argument that contain only numbers and replace them with '*'. The numbers often denote dynamic object IDs (e.g., `5d755f81b0421a673223facdcb7bfecc` in the above log message) that change from execution to execution. Thus, we cannot use these IDs to identify relevant points in a different execution.

An event definition (as in E1 in Figure 9) can be automatically generated from the message components. We assume that we know the type of the node that generated a given log message (e.g., `master` or `regionserver` in HBase). When a node boots up, it is started as one of the possible types for the system, and thus, we would know the type of a node when it is booted up. The entire event definition is not provided for brevity. The other events in the figure are generated from the rest of the log messages in Figure 3. Given event definitions for a sequence of log messages, a tester can inspect the definitions and the buggy scenario, and possibly

```
bug.sc:

E1[node]='regionserver'
E1[stack]=
'org.apache.commons.logging.Log.*
SplitTransaction.createNodeSplitting:
                                 857'

E2[node]='master'
E2[stack]=
'org.apache.commons.logging.Log.*
AssignmentManager.handleRegion: 699'
E2[args]='Handling transition=
RS_ZK_REGION_SPLITTING, server=
d2.nec-labs.com, *,*, region=*'

E3[node]='master'
E3[stack]=
'org.apache.commons.logging.Log.*
ServerShutdownHandler.process: 275'

E4[node]='master'
E4[stack]=
'org.apache.commons.logging.Log.*
AssignentManager.nodeDeleted: 1122'
```

Figure 9. Events in buggy scenario for HBASE-6070 from relevant log messages (Figure 3)

update them based on her knowledge regarding the bug and the system.

4. Evaluation

We have implemented ReproLite for cloud systems written in Java. For a system under test, ReproLite provides executables that can be used as workload for reproducing bugs for that system. For example, for Cassandra [3] (a popular distributed database), we provide scripts that can write data to a given column family in a given keyspace, read data from a given keyspace and column family, drop column family, flush data to disk, etc. For each system event definition in a $\mathbb{RT}$ scenario, ReproLite generates an AspectJ [9] aspect (as explained in Section 3.2.1) that intercepts the given event during execution. The aspects are woven into the given system's bytecode. As the instrumented system executes, its aspects intercept relevant events, and send information regarding those events to $\mathbb{RE}$ (Section 3.2.1) via RPC. $\mathbb{RE}$, which is implemented in Java, decides whether to let the intercepted event proceed or block the event according to the event order in the scenario. External events like network and disk failures are also implemented using AspectJ. Their implementation is described in our previous work, Setsudo [19]. Filtering, concatenation, and diff'ing of logs in $\mathbb{RL}$ are implemented in Python. In all, ReproLite is approximately 4.5K LOC of Java and 1K LOC of Python and shell scripts.

```
bug.sc:
    W1='create-ks.sh test'
    W2='add-cf.sh test cf'
    W3='write.sh 20 test cf'
    W4='flush.sh'
    W5='drop.sh cf'
    W6='add-cf.sh test cf'
    W7='read.sh test cf'

    E2[stack]='BufferedSegmentedFile.
                    getSegment'
    E3[stack]='File.delete'

    W1 * W2 * W3 * W4 *
      (W5 * W6 * W7 || E3 * E2)
```

Figure 10. First attempt at buggy scenario for CASSANDRA-1477

We have experimented ReproLite with two popular cloud databases : Cassandra [3] and HBase [7]. We selected a set of bugs from each system, and used ReproLite to reproduce those bugs. We are interested in answering the following questions: (i) how easy is it to express and validate potential buggy scenarios in ReproLite and eventually find the scenario that causes the bug?, (ii) is ReproLite efficient in reproducing given scenarios?, and (iii) how often do log messages contain the cause of a bug? Our evaluation tries to answer these questions.

4.1 Expressing and validating scenarios in ReproLite

We started out with limited experience regarding the implementation of both Cassandra and HBase, but as we reproduced different bugs for these systems, we found that many a time even if we could not come up with the right set of conditions responsible for a bug in the first attempt, ReproLite helped us to get closer to the cause of the bug by providing us with executions enforcing our initial conjectures that we could inspect and learn from. Being able to quickly express our initial hypotheses and test them helped to follow the right leads and eventually pinpoint the conditions for a bug. We provide our debugging experience in more detail for a couple of bugs in this section.

4.1.1 Reproducing CASSANDRA-1477

Cassandra-1477 [5] is a serious bug (classified as 'blocker') in an older version of Cassandra that can result in loss of data from the database. The bug occurs when a column family is dropped and then added back to a keyspace. Before the column family is dropped, its data is written to disk as SSTable files. The SSTable files are compacted if the number of files is more than the pre-defined number at which compaction should begin. If a SSTable file has been compacted along with other SSTables into a new file, and there is no reference in memory to that SSTable, then that SSTable file would be

```
bug.sc:
    E1[stack]='SSTableDeletingReference.
                    deleteOnCleanup'

    X1 = garbage collection 'cnode'

    W1 * W2 * W3 * W4 *
      (W5 * W6 * W7 ||
         E1 * X1 * E3 * E2)
```

Figure 11. Second (and final) attempt at buggy scenario for CASSANDRA-1477. Missing workload and event definitions are in Figure 10.

deleted eventually. If the file has not yet been deleted, but the column family that it belonged to has been re-created and added to the keyspace, then data written to the column family can be written to that SSTable file. If the SSTable file then gets deleted, all the data written to the column family would be lost.

The report for CASSANDRA-1477 [5] describes two different situations that can arise out of the workload that we have described above (drop a column family from a keyspace, then re-create it and add it back to the keyspace). The second situation has been discussed as being trickier and harder to reproduce, and this is what we aim to reproduce using ReproLite. In fact, one of the developers conjectures the conditions that could be leading to the bug. But, there is no test case that reproduces the bug (the test case provided reproduces the first situation that we do not consider here).

To reproduce the bug using ReproLite, we write down the workload and the events that we are certain should execute for the bug to occur. For example, from the discussion of the bug and the error trace provided in the bug report, we can infer that for the bug to occur, we need a workload that drops a column family from a keyspace, and re-creates and adds it back to the keyspace, and also writes some data to it thereafter. Figure 10 shows a scenario with such a workload. Before dropping the column family, we add enough data to it so that its SSTables are compacted. Also, from the exception trace provided in the bug report, we know that there is a file not found exception when executing `BufferedSegmentedFile.getSegment`. Thus, we can conjecture that a `File.delete` event executes as `BufferedSegmentedFile.getSegment` begins its execution. The deletion event deletes the file being accessed by `BufferedSegmentedFile.getSegment` resulting in the exception. We express our initial conjecture in Figure 10.

When ReproLite enforces the scenario in Figure 10 during execution, we observe that event `E2` (execution of `BufferedSegmentedFile.getSegment`) occurs but not event `E3` (file deletion). From the execution logs and the source code of SSTable

```
bug.sc:
    W1='create-table-cf.sh test cf'
    W2='insert.sh 20 test cf'
    W3='flush.sh test'
    W4='compact.sh test'

    E1[stack]='SplitTransaction.
                    execute'
    E2[stack]='HRegion.
                    compactStores'
    E3[stack]='CatalogJanitor.
                    cleanParent'

    W1 * W2 *
      (W3 * W4 ||
         E1 * E2 * E3 * E2 * E3)
```

Figure 12. Buggy scenario for HBASE-4799

(and related data structures), we can find out that `SSTableDeletingReference.deleteOnCleanup` should execute in order for the timer task that searches for and deletes compacted SSTables without any references to execute. Also, there should be garbage collection to remove dead SSTable in-memory objects before the corresponding SSTable files on disk can be deleted. After adding these events to the scenario in Figure 10, we can reproduce the bug. The final buggy scenario is given in Figure 11.

4.1.2 Reproducing HBASE-4799

HBase-4799 [4] is a critical HBase bug in which even after a parent region has split into two daughter regions, the parent region is not deleted from the system. This happens when the splitting of the daughter regions takes a significant amount of time, and when one of the daughter regions finishes its splitting before the other. If both the daughter regions finish their splitting around the same time, then their references to the parent region are removed together, and the parent region is then deleted from the system. But, if the references to the parent region from one daughter region are removed 'much' before the references from the other daughter region, the parent region is erroneously not deleted from the system.

From the description in the bug report, we can infer that for the bug to happen, splitting of a region should be in progress and the daughter regions should have been created (i.e., `SplitTransaction.execute` should have executed), the references to the parent from one of the daughters should be deleted before the other, and in between the deletion of the references, `CatalogJanitor.checkDaughter` should execute. It took us a couple of attempts to find out that the references to the parent region are deleted in `HRegion.compactStores`. Therefore, `HRegion.compactStores` for one of the daughter regions should finish before that for the other. The final scenario that reproduces the bug is in Figure 12.

System	Bug ID	LOC of scenario	No. of events
Cass	936	10	5
Cass	1353	11	3
Cass	1477	13	5
Cass	3547	6	2
Cass	3862	5	2
Cass	1160	4	3
HBase	3221	7	2
HBase	4395	11	3
HBase	4799	7	3
HBase	6070	8	5
HBase	6050	10	6

Table 1. Complexity of buggy scenarios in $\mathbb{RT}$. Cass stands for Cassandra.

System	Bug ID	Normal exc. time	$\mathbb{RE}$ exc. time	$\mathbb{RE}$ ohd.
Cass	936	25.85	25.88	1
Cass	1353	9	14	1.6
Cass	1477	4	18	4.5
Cass	3547	145	152	1.05
Cass	3862	65.4	65.73	1
Cass	1160	1.339	1.39	1.03
HBase	3221	14	14	1
HBase	4395	9	1200	133
HBase	4799	10	188	18.8
HBase	6070	45	146	3.17
HBase	6050	44	307	6.98

Table 2. Execution overhead of $\mathbb{RE}$ when reproducing bugs

4.1.3 Complexity of buggy scenarios in $\mathbb{RT}$

Table 1 shows the complexity of buggy scenarios for the bugs from Cassandra and HBase. The first column is the name of the system, and the second is the ID of the bug in the Apache bug database [8]. The third column is the lines of code (LOC) of the DSL $\mathbb{RT}$ for expressing the buggy scenarios, and the fourth column is the number of events (system or external) in the buggy scenarios. There are only a few events (≤ 5) involved in each bug, and thus each scenario is small with less than 15 lines of code. Thus, writing scenarios in $\mathbb{RT}$ is generally quick for testers. Even if it may take many iterations (≥ 10) to come up with scenarios for some bugs, we found that the intermediate workloads and schedules helped in understanding what extra constraints were required to reproduce the given bugs.

4.2 Efficiency of ReproLite

For each reproduced bug, Table 2 shows the execution time for executing the workload for that bug without any instrumentation by $\mathbb{RE}$, and with instrumentation and enforcing of the given scenario by $\mathbb{RE}$. The third column in the table is the execution time (in seconds) without using $\mathbb{RE}$, and the fourth column is with using $\mathbb{RE}$. The overhead (time with $\mathbb{RE}$/time without $\mathbb{RE}$) is given in the last column. The last column is in fact an over-approximation of the overhead of $\mathbb{RE}$ since $\mathbb{RE}$ sometimes (e.g., HBase-4395 and HBase-4799) has to wait for operation timeouts (which can be of the order of a few minutes) in order to observe specified events in the given scenario. Thus, even if the overhead of instrumentation might be less, the time taken when executing with $\mathbb{RE}$ might still be a lot more because of the time spent by $\mathbb{RE}$ waiting for timeouts. But, even if the overheads are a coarse over-approximation, the values are close to 1 (meaning almost nil overhead) for half of the bugs. ReproLite has to instrument to observe only the events in the given scenario which are often just a few (less than 10 for our bugs). Hence $\mathbb{RE}$'s instrumentation overhead is low.

System	Bug ID	Cause in logs?	LOC of scenario (gen. from log msgs)
Cass	936	Y	9
Cass	1353	N	N.A.
Cass	1477	Y	12
Cass	3547	N	N.A.
Cass	3862	Y	6
Cass	1160	Y	3
HBase	3221	Y	8
HBase	4395	N	N.A.
HBase	4799	Y	23
HBase	6070	Y	14
HBase	6050	Y	10

Table 3. Automatically generating buggy scenarios from execution logs

4.3 Generating buggy scenarios from execution logs

Table 3 shows if the correct buggy scenarios can be automatically generated by $\mathbb{RL}$ from a set of log messages. The third column in the table indicates if there exist a set of log messages that when executed in the order in which they appear in the erroneous execution (possibly along with external events but without any other ordering constraints on system events) would result in the bug. If yes, then the last column gives the size of the scenario in $\mathbb{RT}$ constructed for the bug from the relevant log messages. We found that correct buggy scenarios could often ($> 70\%$ of the bugs) be constructed from log messages. The $\mathbb{RT}$ scenarios from log messages (Section 3.3) are small and comparable in size to the scenarios manually written by a tester (Table 1).

In order to find the correct buggy scenario from log messages, we always started with a scenario that was generated using log messages in the diff of the erroneous and correct executions. We often had to include a few other log messages into the scenario in order to build the right context to reach the messages already in the scenario. For example, in HBase-4799 (Section 4.1.2), there is a log message (say `L2`) that corresponds to event `E2` (Figure 12) and another message (say `L3`) that corresponds to `E3`. In the correct execution, we would see a sequence of log messages as `L2 L2 L3` (along with many other log messages), and a sequence as `L2 L3 L2 L3` in the erroneous execution. Generating a scenario from the diff of the two executions would generate a scenario from `L3 L2 L3`. But, as we would see from the execution that enforces this scenario that the second `L3` cannot be reached unless there is another `L2`. Thus, we would have to include the first `L2` in the buggy scenario. With increasing understanding of the bug, we could also identify and remove irrelevant log messages from the scenario.

5. Limitations and Threats to Validity

We discuss limitations of our work here.

***Unexpected deadlock*:** The $\mathbb{RT}$ scenarios written by a tester may result in unexpected system deadlock (e.g., if a scenario is incorrect or if the event descriptions are not fine-grained enough to correctly specify the execution points where ReproLite should block). ReproLite waits for a specified amount of time for a scenario to be reproduced. If ReproLite cannot reproduce a scenario, the tester can refine the scenario and run ReproLite again.

***Expressiveness of the DSL*:** $\mathbb{RT}$ has its limitations, and there are scenarios that cannot be expressed in it. For example, we cannot express the specific number of times a method should have executed in an event. Nevertheless, we found $\mathbb{RT}$ in its current form to be expressive enough to quickly prototype the scenarios for various bugs in real-world cloud systems.

***Choice of evaluated bugs*:** For evaluation, we selected bugs with high priority (classification by developers) that we found were hard to reproduce according to the description and discussion in the bug reports. Since we looked at bugs from Cassandra and HBase, ReproLite might not be enough to reproduce bugs from other systems or for other bugs in Cassandra and HBase. However, we believe that ReproLite can be suitably extended (e.g., add suitable constructs to $\mathbb{RT}$) for other systems and classes of bugs.

6. Related work

The work that comes closest to ours is Concurrit [14]. Concurrit enables testers to express a set of thread schedules in a high-level language, and enforces those schedules during execution. However, Concurrit focuses on reproducing concurrency errors in a single process, whereas we focus on system-level concurrency issues. ReproLite enables testers to not only express thread schedules, but also orders of events across different processes and external events like network failures and garbage collection. Workloads (e.g., client reads and writes from a distributed database) can also be interleaved with system and external events. Moreover, Concurrit relies on developers' insights to reproduce a failure. ReproLite provides assistance to understand the cause of a failure from logs, and automatically creates a scenario in its DSL from a sequence of log messages.

Most of the previous work [16, 24, 25, 27] on debugging distributed systems has focused on building tools that can capture and record enough information about non-determinism in a system execution (e.g., races in shared variable accesses, order in which network messages are sent and received, OS signals, preemptions, etc.) that would allow the tools to deterministically replay the execution later. The more non-deterministic choices or events that are logged, the higher are the chances to replay the exact execution observed. But, logging involves a significant overhead, and system developers and testers often do not want to incur this overhead by aggressively logging all non-determinism during execution. Thus, instead of increasing logging for a system, ReproLite enables testers to easily and quickly express hypotheses that they form from incomplete logs, and validate those hypotheses with little execution overhead.

There has been a lot of work on providing high-level languages to enable testers to query logs during debugging [21], and also to express distributed predicates and conditions to test against [15, 23, 24]. In recon [21], testers can write SQL-like queries to extract specific events out of logs (e.g., events involving specific nodes or communication channels). The DSL in ReproLite ($\mathbb{RT}$) also allows testers to identify events involving specific nodes or communication links. Distributed predicates are complimentary to ReproLite– the predicates can help testers to find subtle bugs during execution, and the testers can then use the execution logs to form and validate conjectures regarding the cause of the bugs.

There have been static analysis based approaches to finding the cause of an error using logs. For instance, by using path- and context- sensitive program analysis and constraint satisfiability solving, Sherlog [26] cross-checks logs from a failed execution and source code to obtain a path slice for a given bug. However, Sherlog is geared towards sequential programs. Adaptation of Sherlog to distributed systems would be complimentary to ReproLite and enhance $\mathbb{RL}$.

Black-box stress testing (e.g., HPs Load Runner [11] and Unified Functional Testing [12], Apache JMeter [10], and Selenium [13]) is popular in the industry for testing large-scale systems. Stress testing involves repeated execution of unit-tests or system-tests (e.g. 1,000 repeated runs) to find bugs. Bugs in large-scale distributed systems often occur under specific workloads and event orders that are uncommon. Even if such a bug is found during stress testing, it would be hard to reproduce the bug in subsequent executions. ReproLite enables a tester to repeatedly reproduce a hard bug. The $\mathbb{RT}$ scenario for the bug also serves as a regression test for the bug.

Acknowledgments

We thank Yannis Smaragdakis and Christoph Reichenbach for helpful comments and interesting discussions.

References

[1] Am.nodedeleted and ssh races creating problems for regions under split. `https://issues.apache.org/jira/browse/HBASE-6070`.

[2] Apache ZooKeeper. `http://zookeeper.apache.org`.

[3] Cassandra. `http://cassandra.apache.org/`.

[4] Catalog janitor logic bug causes region leackage. `https://issues.apache.org/jira/browse/HBASE-4799`.

[5] drop/recreate column family race condition. `https://issues.apache.org/jira/browse/CASSANDRA-1477`.

[6] Hadoop. `http://hadoop.apache.org/`.

[7] HBase. `http://hbase.apache.org/`.

[8] System dashboard - ASF JIRA. `https://issues.apache.org/jira`.

[9] The Aspectj Project. `http://www.eclipse.org/aspectj/`.

[10] Apache jmeter. `http://jmeter.apache.org/`, July 2014.

[11] Hp - load runner. `http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/`, July 2014.

[12] Hp - unified functional testing. `http://www8.hp.com/us/en/software-solutions/unified-functional-testing-automation/`, July 2014.

[13] Selenium automates browsers. `http://www.seleniumhq.org/`, July 2014.

[14] T. Elmas, J. Burnim, G. Necula, and K. Sen. Concurrit: A domain specific language for reproducing concurrency bugs. PLDI, 2013.

[15] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. NSDI, 2007.

[16] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. ATC, 2006.

[17] 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 *NSDI*, 2011.

[18] 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 *ACM Symposium on Cloud Computing (SOCC) (to appear)*, 2014.

[19] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N. Papakonstantinou. Setsudo: Perturbation-based testing framework for scalable distributed systems. *TRIOS: Conference on Timely Results in Operating Systems*, 2013.

[20] P. Joshi, H. S. Gunawi, and K. Sen. PREFAIL: A programmable tool for multiple-failure injection. In *OOPSLA*, pages 171–188. ACM, 2011.

[21] K. H. Lee, N. Sumner, X. Zhang, and P. Eugster. Unified debugging of distributed systems with recon. DSN, 2011.

[22] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, Broomfield, CO, Oct. 2014. USENIX Association.

[23] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3s: Debugging deployed distributed systems. NSDI, 2008.

[24] X. Liu, W. Lin, A. Pan, and Z. Zhang. Wids checker: Combating bugs in distributed systems. NSDI, 2007.

[25] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the Euromicro Conference on Real-time Systems*, ECRTS, 2000.

[26] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from runtime logs. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 143–154. ACM, 2010.

[27] C. Zamfir, G. Altekar, and I. Stoica. Automating the debugging of datacenter applications with adda. DSN, 2013.