DFix: Automatically Fixing Timing Bugs in Distributed Systems

Guangpu Li
University of Chicago
USA
cstjygpl@uchicago.edu

Haopeng Liu
University of Chicago
USA
haopliu@uchicago.edu

Xianglan Chen*
University of Sci. and Tech. of China
China
xlanchen@ustc.edu.cn

Haryadi S. Gunawi
University of Chicago
USA
haryadi@cs.uchicago.edu

Shan Lu
University of Chicago
USA
shanlu@uchicago.edu

Abstract

Distributed systems nowadays are the backbone of computing society, and are expected to have high availability. Unfortunately, distributed timing bugs, a type of bugs triggered by non-deterministic timing of messages and node crashes, widely exist. They lead to many production-run failures, and are difficult to reason about and patch. Although recently proposed techniques can automatically detect these bugs, how to automatically and correctly fix them still remains as an open problem. This paper presents DFix, a tool that automatically processes distributed timing bug reports, statically analyzes the buggy system, and produces patches. Our evaluation shows that DFix is effective in fixing real-world distributed timing bugs.

CCS Concepts • **Software and its engineering** → **Cloud computing**; *Software maintenance tools*; • **Computer systems organization** → Reliability.

Keywords Distributed system; Timing; Bug fixing

ACM Reference Format:
Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S. Gunawi, and Shan Lu. 2019. DFix: Automatically Fixing Timing Bugs in Distributed Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314620

*The work was done when Xianglan Chen visited University of Chicago

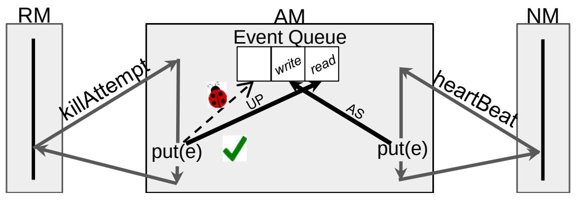


Figure 1. A message-timing bug in MapReduce[3].

1 Introduction

Distributed systems such as scale-out storage systems [20, 24, 28, 57] and cloud computing frameworks [23, 56] are the backbone of our computing ecosystem. High availability of these systems is crucial, with minutes of outage costing millions of dollars [55, 66], but severely threatened by software bugs, particularly distributed timing bugs [31, 46, 70], a type of bugs triggered by non-deterministic timing of message communication or component failures (e.g., node crashes).

Distributed timing bugs impose a particularly large threat to system availability for several reasons. They are difficult to expose before code release, given their non-deterministic nature, and the limited scale and duration of in-house testing. Consequently, they widely exist in the field [14, 31, 33, 46] and easily manifest during large-scale and long-running production deployment, contributing to more than a quarter of cloud-system failures [70]. Although many recent tools [25, 32, 43, 45, 47–49, 63] can automatically detect these bugs, system availability does not improve until after these bugs are fixed. Unfortunately, correctly fixing distributed timing bugs is challenging for developers, as it involves global reasoning beyond one thread or one node, and often requires non-traditional synchronization, as we will elaborate below.

1.1 Examples

There are two types of distributed timing bugs: message-timing bugs and fault-timing bugs. Figure 1 illustrates a *message-timing bug* in MapReduce, triggered when a user kills a job unexpectedly early. From the figure, we can see two series of concurrent operations involved in this bug. Starting

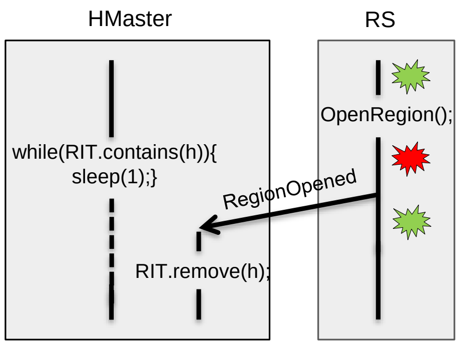


Figure 2. A fault-timing bug in HBase [2] (red explosion: bug-triggering crash; green explosion: tolerable crash).

from the left-hand side of the figure, when a user issues a task-kill command, the resource-manager RM sends an RPC `killAttempt` to the application-manager AM; the RPC handler in AM then creates an event `UP` to asynchronously check the status of job J. Starting from the right-hand side of the figure, node-manager NM sends an RPC `heartBeat` to AM; the corresponding RPC handler in AM then creates an event `AS` to asynchronously update J's status to `ASSIGNED`. Since these two sets of operations are concurrent with each other, `UP` may non-deterministically read the job status *before* `AS` updates the status to `ASSIGNED`, causing a `JobNotExisting` exception and a job abort.

A naïve way to fix this bug is to use a single-machine synchronization primitive like condition-variable `wait` to force the status read in `UP` to wait for the status update in `AS`. However, this could cause safety and liveness problems. The blocking wait may cause an RPC *timeout* in the RPC client, potentially leading to failures. Furthermore, it may cause deadlocks by blocking not only the status read but also the status update, as these two accesses may share the same event-handling or RPC-handling thread — the number of event-queue or RPC-server handling threads is usually configurable and is 1 by default in many of our benchmarks.

Fixing this bug in a distributed manner is still challenging. First, reasoning about the timing relationship among distributed operations is error prone. For example, naïvely forcing RM to send `killAttempt` after NM sends `heartBeat` can*not* guarantee event `UP` to execute after event `AS` if the system configures multiple event or RPC handling threads. Second, it is not trivial to enforce a distributed execution order. It often involves adding either a distributed synchronization service like ZooKeeper [4] or a few new RPCs/messages among various nodes for coordination. Both may produce patches that are too complicated to adopt.

Figure 2 illustrates a *fault-timing bug* in HBase, triggered by an unexpected region-server (RS) node crash in the middle of opening a region. In HBase, a region server *RS* opens a region by executing an `OpenRegion` function, illustrated in the figure. This function first remotely causes a record to be inserted into HMaster's region-in-transition `RIT` map, indicating that a region is to be opened, and later remotely removes this record from `RIT` through a `RegionOpened` message, indicating that the region has been opened. If *RS* happens to crash in between, denoted by the red-explosion symbol in figure 2, HMaster would be stuck in a loop `while (RIT.contains (h))`, waiting for the record to be removed forever and making the whole HMaster service unavailable.

Clearly, fixing this bug cannot rely on traditional wait-based synchronization primitives like locks, condition variables, and semaphores — it is useless to wait as one cannot predict when a node would crash. Traditional file-system crash consistency does not help here either, as the impact of the crash goes beyond one node and beyond file systems.

1.2 Challenges & Goals

Those challenges discussed above widely apply:

- Fixing distributed timing bugs often cannot rely on traditional synchronization primitives like locks and condition variables, as blocking waits cannot help fix fault-timing bugs, and can introduce new bugs if put in event/RPC handlers where most message-timing bugs are located [46, 48].
- Fixing distributed timing bugs often requires global code changes and reasoning about operation timing and side effects across threads, processes, and nodes.

In addition to these unique challenges, there are also generic challenges like keeping patches not only correct, but also reasonably simple and well performing [64].

These challenges are reflected in how developers fix distributed timing bugs in practice.

Fixing distributed-timing bugs lacks dominant or systematic strategy in practice. According to the previous study [46], a variety of schemes, including ad-hoc ones, are used by developers: about half are fixed by a set of schemes like message retries, message ignoring, message faking (e.g., the bug in Figure 2), and others that allow software to better tolerate buggy timing; <30% are fixed by proactively disabling buggy timing, including <10% using distributed waits; about 20% are fixed through significant data-structure and semantic changes (e.g., the bug in Figure 1), partly due to the difficulty of finding semantic-preserving patches.

Fixing distributed-timing bugs is time consuming and error prone. Similar to single-machine timing bugs [29], distributed timing bugs take days to months to fix, with bug understanding and patch design all taking time. Even distributed timing bugs tagged with the highest priority [1, 5] suffer from incorrect patch releases after many days' patch design and review.

Although techniques have been proposed to help detect [32, 43, 45, 47–49, 63] and diagnose [21, 26, 42, 51, 54, 70, 71] distributed timing bugs, no techniques have been proposed to help automatically fix them. Auto-fixing has been proposed and heavily researched for local timing bugs where some use locks, condition variables, or well designed waits to fix multi-threaded concurrency bugs after these bugs are detected [36, 38–40, 50, 53, 67], and some [11] use domain-specific event policies to fix event races in web applications. Unfortunately, these techniques cannot handle distributed timing bug challenges discussed above.

Overall, it is desirable to have techniques that automatically apply a unified strategy to fix many distributed-timing bugs through best-effort patches similar to what a human would create. Such a technique can save manual effort in fix-strategy design, global reasoning, global code changes, and code correctness review. The resulting patches could serve as either temporary patches, which improve system availability while helping developers to figure out final patches (as long as the automatically generated patches are in source code, which DFix patches are), or directly as final patches.

1.3 Contributions

This paper proposes DFix, a tool that takes in distributed timing bug reports, and automatically generates best-effort patches for many of them, through static program analysis.

At the high level, DFix does not use traditional blocking-wait synchronization primitives. Instead, it systematically generates patches that handle observed buggy timing through rollbacks [40, 58, 59, 62, 67] or fast-forwards.

Specifically, to fix a message-timing bug, DFix patches a selected code region r to observe if r's thread or node is executing undesirably fast, and if so, slow down by repeated roll-back and re-execution.

To fix a fault-timing bug, DFix patches a selected region r to observe whether another node has crashed at an undesirable moment and if so, rollback or fast-forward selected operations on behalf of the crashed node, pretending the crash occurred earlier or later.

At the low level, DFix uses static analysis to automatically decide where and how to observe buggy timing, and where and how to conduct rollback or fast-forward, so that the resulting patches satisfy several properties:

1. They are designed to only constrain the timing and not to change any computation logic of the original software execution. Specifically, the static analysis of DFix carefully avoids liveness violations like inserting blocking-waits inside RPC/event handlers, and safety violations like re-executing non-idempotent operations, etc.
2. They cover a wide variety of real-world fixing schemes through a unified high-level strategy. Depending on the rollback/fast-forward region location and length, some DFix patches are essentially equivalent with developer patches that proactively disable buggy timing, and some are equivalent with developer patches that reactively tolerate buggy timing (e.g., in the case of Figure 2); some involve message retries, and some essentially ignore or fake messages.
3. They often involve multiple threads and nodes. Through automated analysis and transformation, DFix relieves developers from distributed reasoning and code changes.
4. They do not introduce severe degradation to performance or code complexity. DFix intentionally gives up its bug fixing if the patch would be too complicated.

We evaluate DFix on *all* the 22 real-world distributed timing bugs reported by recently proposed message-timing bug detector DCatch [48] (10 bugs) and fault-timing bug detector FCatch [49] (12 bugs), which come from Cassandra, HBase, MapReduce, and ZooKeeper. DFix automatically fixes 17 of them with similar performance and simplicity as manual patches. The source code of DFix and details about all the bug benchmarks, including patches generated by DFix and patches provided by developers, are all available at our website https://github.com/SpectrumLi/TimingBugFixing.

In summary, DFix is not a panacea. There are distributed timing bugs that DFix cannot fix. There are also bugs that DFix can fix but cannot fix in the most desirable way, particularly when the ideal patch requires semantics changes, which DFix has no conceivable way to automatically generate with correctness guarantees. However, we believe DFix provides a solid starting point towards solving this critical problem of patching real-world distributed timing bugs, which is very challenging for developers to manually reason about even *after* bug detection as we will see in DFix design, and improving availability of distributed systems.

2 Background

Distributed Timing Bugs Previous study [46] categorizes real-world distributed timing bugs into two types, with each type containing two sub-categories, as illustrated in Figure 3. Before we present how DFix fixes these two major types in Section 3 and 4, we explain all the sub-categories below.

Message-timing bugs can be categorized into order violations and atomicity violations. A *message order violation* manifests when an operation B unexpectedly accesses a shared resource before an operation A, like reading data before initialization (figure 3a). A or B or both are initiated through messages. The bug shown in figure 1 belongs to this category. A *message atomicity violation* manifests when an operation B unexpectedly executes in between and hence violates the atomicity of a code region A_1–A_2 (figure 3b). A_1, A_2, and B access the same resource, with at least one initiated by messages.

Fault-timing bugs can be similarly categorized. In a *fault order violation*, operation B cannot proceed until A initiated by another node has executed. Unfortunately, when that other node crashes before A or drops a message whose handler conducts A, B waits forever and causes the system to

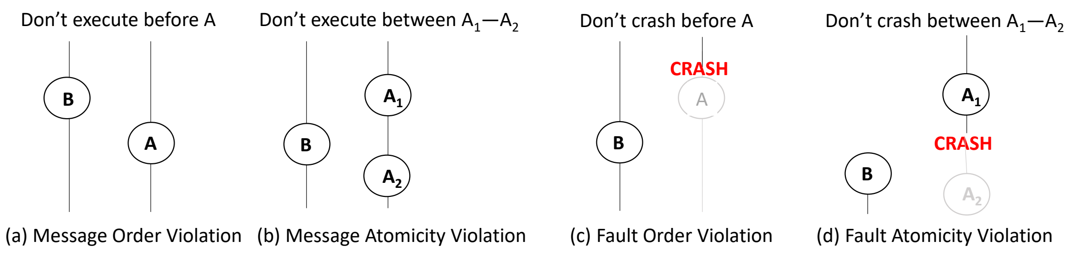

Don't execute before A
B
A
(a) Message Order Violation
Don't execute between A1—A2
B
A1
A2
(b) Message Atomicity Violation
Don't crash before A
CRASH
A
B
(c) Fault Order Violation
Don't crash between A1—A2
A1
CRASH
B
A2
(d) Fault Atomicity Violation

Figure 3. Distributed message-timing bugs ((a) and (b)) and distributed fault-timing bugs ((c) and (d)).

partially or completely hang (figure 3c). The bug shown in figure 2 belongs to this category. In a *fault atomicity violation*, when a node unexpectedly crashes in a specific time window, denoted by A_1–A_2 in figure 3d, it leaves a system state that cannot be correctly handled by its recovery routine. Once that system state is read by the operation B in figure 3d, the recovery attempt or the whole system fails.

DFix Front End As an auto-fixing, *not* bug-detection, tool, DFix relies on its front end to provide accurate bug reports. Ideally, a bug report should contain (1) the bug type — which one out of the four types listed in figure 3; (2) racing instructions and buggy window boundaries — all the operations illustrated in figure 3, each identified by its static instruction ID and dynamic context (call stack and causality stack), under which the bug is triggered. DFix is *not* responsible for fixing unreported bugs and unreported contexts.

Here, *causality stack* refers to a sequence of causal operations that lead to the execution of I: when I is inside an event handler, the top of its causal stack is the corresponding event enqueue operation; when I is inside a message or RPC handler, the top of its causality stack is the message/RPC sending operation from another node; when I is inside a regular thread, the top is the thread creation from its parent. Causality stack is crucial to identify dynamic context related to bug manifestation [48, 49].

We expect DFix to work with most dynamic distributed-timing-bug detectors, as information like bug type and racing instructions are fundamental bug components and should be reported by most detectors. The dynamic context needed by DFix may not be reported by default, but should be easy to extract from any dynamic detector. The current DFix prototype uses DCatch [48] and FCatch [49] as front end for message-timing and fault-timing bugs, respectively. The bug report inputs used in DFix experiments can be found at https://github.com/SpectrumLi/TimingBugFixing.

3 Fixing Message-Timing Bugs

3.1 Overview

This section uses a simplified example from the real-world bug in figure 1 to explain how a DFix patch works and what are the key challenges in patch generation.

Running Example Figure 4 contains data races to a shared heap object `state` from the event handler `handle` and the RPC function `update`, denoted as A and B respectively. A job fails if B reads `state` before A updates `state`. Consequently, the patch should make sure A executes before B. Note that, B exists in an event handler instead of an RPC function in the real bug illustrated in figure 1. The simplification in figure 4 does not make the bug easy to fix: forcing RM to invoke RPC after NM (not shown in figure 4) still cannot guarantee B to execute after A due to the asynchronous nature of A; blocking-wait before B still incurs deadlocks and undesirable RPC timeouts. We will explain how to fix the original bug later in this section.

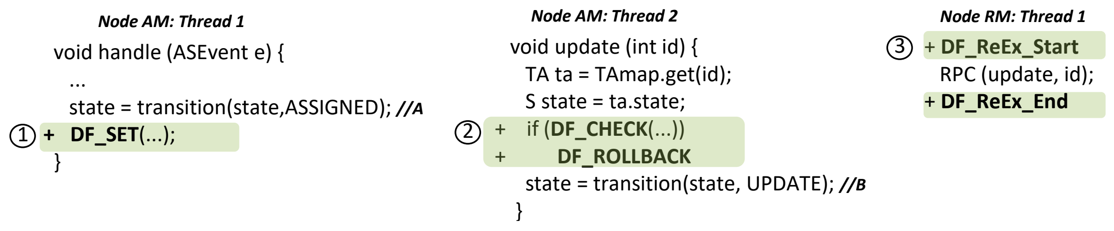
Roadmap All DFix patches for message-timing bugs, including both order violations and atomicity violations, contain similar components: somewhere before B like ② in figure 4, the patch calls `DF_CHECK` to see whether B's thread is executing too fast based on a flag set by `DF_SET` somewhere else like ① in figure 4, and, if so, the patch calls `DF_ROLLBACK` to roll back the execution to a place marked by `DF_ReEx_Start` like ③ in figure 4 and uses repeated re-execution from `DF_ReEx_Start` to slow down (the meaning of B is explained in Section 2).

To generate these patches, DFix takes several steps as sketched out in Algorithm 1:

First (line 2), DFix decides the re-execution region location by analyzing functions on the causality stack of B. The starting location of re-execution is particularly important, which is denoted as $\mathrm{Loc}_{\mathrm{Start}}$ in Algorithm 1 and exemplified by location ③ in figure 4. The details of this step is explained in Section 3.2.

Second (line 5), DFix decides where to check the buggy-timing and potentially initiate rollback by analyzing functions on the causality stack of B and the $\mathrm{Loc}_{\mathrm{Start}}$ identified above. This location is denoted as $\mathrm{Loc}_{\mathrm{CHECK}}$ in Algorithm 1. It is sometimes right before the race instruction like ② in figure 4, but not always. The details are in Section 3.3.

Third (line 8), DFix generates code snippets that conduct the rollback by analyzing the locations of `DF_ReEx_Start` and `DF_CHECK` — whether they are inside the same function, whether they are inside the same node, and others. These code snippets are underneath `DF_ReEx_Start`, `DF_ReEx_End`,



```
Node AM: Thread 1
    void handle (ASEvent e) {
      ...
      state = transition(state,ASSIGNED); //A
①  +  DF_SET(...);
    }

Node AM: Thread 2
    void update (int id) {
      TA ta = TAmap.get(id);
      S state = ta.state;
②  +  if (DF_CHECK(...))
   +      DF_ROLLBACK
      state = transition(state, UPDATE); //B
    }

Node RM: Thread 1
③  + DF_ReEx_Start
     RPC (update, id);
   + DF_ReEx_End
```

Figure 4. DFix patch for a bug simplified from figure 1 (green part illustrates the patch; identities of A and B, including their bug-triggering causality stacks, are the inputs to DFix).

and `DF_ROLLBACK` located at ② of figure 4. The details are explained in Section 3.4.

Finally, DFix generates code snippets to allow its patch finding out the buggy timing at run time. Specifically, DFix decides where to set and unset a buggy-timing flag (line 11–15 in Algorithm 1), exemplified by the `DF_SET` located at ① in figure 4; DFix also generates the parameters for `DF_SET`, `DF_UNSET`, and `DF_CHECK` calls (line 18–21, 27–31). The details are presented in Section 3.5.

3.2 Where does the Re-execution Start?

The starting point of re-execution (i.e., the rollback destination) cannot be arbitrary. Particularly, it cannot be inside resource-holding or time-sensitive regions, like lock critical sections or event/RPC handlers, as repeated re-executions can cause deadlocks and/or timeouts. In fact, there may be no suitable rollback destination throughout the thread of B, because if B is in an RPC or event handler, everywhere in B thread is prone to deadlocks and/or timeouts.

Solutions Starting from the location right before B, DFix searches backward for a suitable rollback destination along B's causality stack (i.e., *not* call stack like previous local concurrency-bug fixing tools [38, 39]). Given a location L, DFix makes the following check: (1) if L is inside a lock critical section, the search continues at where the corresponding lock was acquired; (2) if L is inside an event handler, the search continues at where the corresponding event was enqueued in another thread; (3) if L is inside an RPC or message handler, the search continues at where the RPC request was invoked at another node. If L does not fall into any of the three cases above, it is chosen as the starting point of re-execution (i.e., rollback destination), denoted as $\mathtt{Loc}_{\mathtt{Start}}$.

For the bug in figure 4, the search starts from inside the `update` RPC handler, right before B in the middle column of the figure. Since the rollback destination cannot be inside an RPC handler, the search moves to the RPC caller side on node RM, and ends at right before the RPC request on RM (i.e., ③ in figure 4).

[1]Inputs to this algorithm are provided by the front-end bug detector, and are explained in Section 2 and Figure 3. We use `PATCH X @ Y` to indicate that a DFix patch inserts code snippet `X` at location `Y`.

Algorithm 1: Patching a message-timing bug[1].

```
Input    : {$A$, $B$} for an order violation,
           {$A_1$, $A_2$, $B$} for an atomicity violation
1   /*locate the re-execution region so that re-execution is not inside
      lock critical sections or RPC handlers */
2   {$Loc_{Start}$, $Loc_{End}$} = FindReexecutionRegion (B);
3
4   /*locate the check location so that re-execution is idempotent*/
5   $Loc_{CHECK}$ = FindCheckLocation ($Loc_{Start}$, $B$);
6
7   /*generate rollback routine based on where re-execution region is*/
8   {DF_ROLLBACK, DF_ReEx_Start, DF_ReEx_End}=
      CausalityRollBack ($Loc_{Start}$, $Loc_{CHECK}$);
9
10  /*locate flag (un)set location to observe buggy timing*/
11  if (the bug is an Order-Violation) then
12      $Loc_{SET}$ = After (A);
13  else
14      $Loc_{SET}$ = Before ($A_1$); $Loc_{UNS}$ = After ($A_2$);
15  end
16
17  /*pre-compute the race-location for DF_CHECK*/
18  $RaceID_{pre}$ = Slicing ($B$.raceobj, $Loc_{CHECK}$);
19  if ( !Idempotent ($RaceID_{pre}$) ) then
20      $RaceID_{pre}$ = Constant;
21  end
22
23  /* Generate the Patch */
24  PATCH (
25    DF_ReEx_Start                @ $Loc_{Start}$,
26    DF_ReEx_End                  @ $Loc_{End}$,
27    DF_SET (ID(A.raceobj))       @ $Loc_{SET}$, //order vio.
28    DF_SET (ID($A_1$.raceobj))     @ $Loc_{SET}$, //atom. vio.
29    DF_UNSET (ID($A_2$.raceobj)) @ $Loc_{UNS}$, //atom. vio.
30    "IF(DF_CHECK($RaceID_{pre}$)) {DF_ROLLBACK}"
31                                 @ $Loc_{CHECK}$);
```

3.3 Where to Check Timing and Initiate Rollback?

Somewhere before B and after the staring point of re-execution $\mathtt{Loc}_{\mathtt{Start}}$, the patch should check for the buggy timing and potentially initiate the rollback (i.e., invoking DF_ROLLBACK based on the DF_CHECK result in figure 4). There is a trade-off here about the DF_CHECK and hence DF_ROLLBACK location: on one hand, we prefer the checking to be closer to $\mathtt{Loc}_{\mathtt{Start}}$, as a shorter rollback/re-execution region is easier to guarantee correctness; on the other hand, we prefer the checking to be closer to B, as it is easier to predict whether B will execute shortly before B.

Solutions To balance this tradeoff, DFix starts from $\mathtt{Loc}_{\mathtt{Start}}$ identified earlier, and searches forward along B's causality chain towards B for a suitable location $\mathtt{Loc}_{\mathrm{CHECK}}$, so that $\mathtt{Loc}_{\mathrm{CHECK}}$ dominates B and the resulting region is the longest one that contains *no side-effect* instructions. This way, we can easily guarantee that repeated rollback and re-execution will not change program semantics without checkpoints. DFix gives up its patching attempt if $\mathtt{Loc}_{\mathrm{CHECK}}$ ends up in a different node from B.

Naturally, all read operations and writes to stack variables whose stack frames are rolled back (e.g., the write to ta and state in function update of figure 4) are free of side effects. An event enqueue, an RPC call, and a message sending are free of side effects if the corresponding handler only updates return values or stack variables during re-execution. This applies for RM's RPC request in figure 4, because the update function only updates stack variables before B.

3.4 How to Rollback

Since every rollback region in DFix patches is free of side effect, rollback and re-execution do not require any checkpointing and can be repeated for many times without introducing new bugs. We just need to pick different re-execution mechanisms for intra-node situations and inter-node situations, respectively.

To roll back a code region inside one thread or one handler, the patch puts the code into a loop and rolls back by a loop continue — such a loop and loop continue are put underneath DF_ReEx_Start and DF_ReEx_End surrounding the RPC call on RM in figure 4. Inter-procedural rollback is done similarly and uses exception throw to rollback from the callee function to the caller.

To roll back code regions across threads, handlers, or nodes, the rollback procedure is essentially multiple intra-thread/handler rollbacks chained together. Starting from the thread/handler where rollback is initiated (e.g., DF_ROLLBACK in figure 4), every thread/handler simply terminates itself and then informs its causality parent (i.e., the parent thread, the message sender, the event creator, etc.) to also rollback until the rollback-destination thread/handler is reached, where a loop continues to launch re-execution (e.g., the loop underneath DF_ReEx_Start and DF_ReEx_End in figure 4).

How to inform the causality parent to rollback varies for different causality operations:

For RPC calls or other synchronous messages, this is done by throwing a remote exception/error. For example, in figure 4, DF_ROLLBACK throws an exception with the exception handler executing loop continue on RM as part of DF_ReEx_End.

For asynchronous causal operations like event enqueues, a flag variable is used to coordinate between the event handler and the event creator function C. After C enqueues the corresponding event, it repeatedly checks the flag until: (1) the flag is set by the event handler indicating that no buggy timing is observed and hence no rollback is needed, in which case C continues its execution; (2) the flag is set by the event handler indicating that buggy timing is observed and rollback is needed, in which case C rolls back itself and notifies its causality parent; or (3) the flag is not updated for a threshold amount of time and C is a time-sensitive code region like in an RPC handler. In this case, C rolls back and throws an exception to trigger an RPC resending. The patch makes sure that the re-sended RPC does not cause redundant event enqueues. This is how the original bug in figure 1 is fixed in a semantics-preserving way.

3.5 How to Observe the Buggy Timing

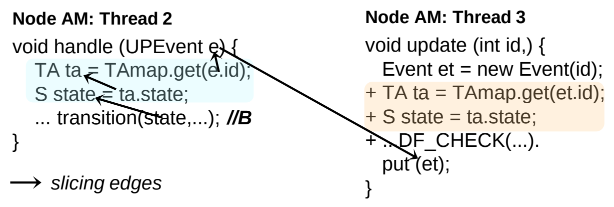
At the checking site identified earlier, the DFix patch checks a flag to decide whether the current thread is executing too fast. In figure 4, this checking is done by DF_CHECK function and the function parameter helps decide which flag to check.

The challenge here is to maintain a precise flag, particularly (1) when to set and unset the flag, and (2) how to distinguish flags that belong to different dynamic instances of one static bug. The latter was ignored by local concurrency-bug fixing tools [38, 39], but is particularly important in cloud systems, where the same instruction could execute for many times (e.g., once per task attempt or job) and ordering is expected only among certain dynamic instances (e.g., the status read of job J needs to wait for initialization of J only, not other jobs).

Solutions For an order violation $\{A, B\}$, a flag is set right after A, and B can execute once the flag is set (line 12 in Algorithm 1). For an atomicity violation $\{A_1–A_2, B\}$, a flag is set right before A_1 and reset right after A_2, and B can execute if the flag is not set (line 14 in Algorithm 1).

Since different dynamic instances of A or B could access different objects, DFix creates a flag map for each bug, indexed by the hash-code of the race object m accessed by racing instructions. DF_CHECK and DF_SET in figure 4 use such hash code as parameters, which help locate the right flag.

The remaining question is to figure out which object will be accessed by the buggy instruction B at the checking site. This is trivial if DF_CHECK is right before B like that in figure 4, but challenging otherwise like that in figure 5.



```
Node AM: Thread 2
void handle (UPEvent e) {
   TA ta = TAmap.get(e.id);
   S state = ta.state;
   ... transition(state,...); //B
}
```

```
Node AM: Thread 3
void update (int id,) {
   Event et = new Event(id);
+ TA ta = TAmap.get(et.id);
+ S state = ta.state;
+ ... DF_CHECK(...).
   put (et);
}
```

Figure 5. Pre-computation of race-object hash-code.

DFix first statically identifies all instructions that may affect the race-object identity along any path between `DF_CHECK` and *B*. To do so, DFix extends traditional static slicing by concatenating slices in multiple threads or handlers together along the causality chain (line 18 in Algorithm 1). DFix then puts these instructions together, with variable names adjusted (e.g., replace `this.f1` by `o.f1`) or replaced by corresponding getter method (e.g., replace `this.f1` by `o.getf1()`), to get the race-object reference for `DF_CHECK`. Finally, DFix statically checks whether the above pre-computation is idempotent. If not, DFix simply uses a constant number as the flag-map key, making all dynamic instances of a bug share the same flag (line 19–21 in Algorithm 1).

For example, in figure 5, DFix wants to pre-compute the race-object identity, which is `state` used by *B* in Thread 2, right after the event-enqueue in Thread 3. To accomplish this, static slicing starts from *B* and identifies two operations that determine the identity of `state` in the `handle` function, as indicated by the arrows in figure 5. Then, following the causality chain, DFix knows that the `handle`'s parameter `e` is actually the local variable `et` inside the `update` function in Thread 3. Then, these related instructions are cloned to function `update` with some name adjustment (i.e., replacing `e` by `et`) to pre-compute the race-object identity.

The above pre-computation is accurate when other threads cannot affect which object *B* would access between `DF_CHECK` and *B* (i.e., no time-of-check time-of-use races [15, 16]).

DFix proves this assumption when (1) *B* accesses a global/static object; or (2) the pre-computation only reads variables whose content cannot be changed by other threads between the checking and *B*, like local variables, final variables, heap objects whose references haven't been put to another shared objects yet. etc. When DFix cannot prove so, it still generates the patch. In the very rare cases when a TOCTOU race happens right before a buggy timing occurs, the patch would fail to fix the bug but would not introduce new bugs.

3.6 Patch Correctness Analysis

DFix carefully prevents its patches from introducing new bugs through several efforts. First, DFix makes sure that the rollback/re-execution region in its patch does not contain any side-effect operations (Section 3.3), and the extra pre-computation conducted to observe the buggy timing is idempotent (Section 3.5). This way, DFix patches guarantee to only change the execution timing, but not introducing new program semantics. Second, a DFix patch does not introduce any blocking wait inside RPC handlers, given its overall design and its safety-net timeout (Section 3.4), and hence does not introduce any unexpected RPC timeout exceptions. Third, DFix sets a threshold for the number of rollbacks in its patch (20 by default). This way, a DFix patch is guaranteed not to cause deadlocks, although in theory it may fail to prevent a bug manifestation that could have been prevented if there were more rollbacks. We further elaborate on the last point. Naïvely, one may want to delay *B* until *A* is executed. However, doing so may cause deadlocks if *A* never executes, a realistic issue in distributed systems where a node crash easily makes an instruction that was supposed to execute disappear. This is why DFix needs to set a threshold for the number of rollbacks.

Furthermore, although unnecessary for patch-correctness guarantees, DFix still makes its best effort in analyzing whether the waited-for instruction is guaranteed to execute, if there is no node crashes or message drops, to provide more information to developers. To do so, DFix leverages the traditional dominating and post-dominating relationship analysis [12] and also extends it with causality information to reason beyond one thread or one node — if a causality child like an RPC handler has executed, we know for sure that the causality parent like the RPC caller has executed. This way, DFix can ensure that the waited-for instruction is guaranteed to execute for most message-timing bugs that we encountered in fault-free situation (Section 6.3).

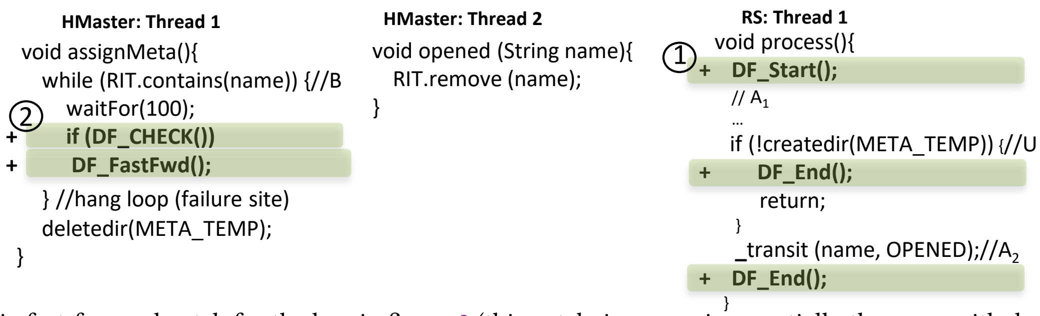
4 Fixing Fault-Timing Bugs

4.1 Overview

We use the bug discussed in figure 2 to demonstrate how DFix fixes a fault-timing bug through fast-forward.

Running example As shown in figure 6, HMaster's thread 1 executes a loop that exits only when a specific entry `name` is removed from the `RIT` map. This entry is inserted through a request by *RS* right before the `process` function, and is removed through an `OPENED` handler in HMaster-Thread 2 also remotely requested by *RS* at A_2. When *RS* crashes inside this `process` function before the `OPENED` request is sent out at A_2, HMaster gets stuck in the while loop, making the whole HBase system unavailable.

Roadmap All DFix patches for fault-timing bugs contain similar components: right before where the failure gets exposed according to the bug report like ② in figure 6, the patch checks whether another node N_{fault} has crashed in the buggy time window. If so, this checking node attempts to either rollback (in rollback patch) or fast-forward (in fast-forward patch like figure 6) some of the crash-persistent operations on behalf of N_{fault} (i.e., inside `DF_FastFwd` in figure 6); logging surrounding the buggy time window (i.e.,



```
HMaster: Thread 1
void assignMeta(){
  while (RIT.contains(name)) {//B
    waitFor(100);
②
+   if (DF_CHECK())
+     DF_FastFwd();
  } //hang loop (failure site)
  deletedir(META_TEMP);
}
```

```
HMaster: Thread 2
void opened (String name){
  RIT.remove (name);
}
```

```
RS: Thread 1
  void process(){
① + DF_Start();
    // A1
    ...
    if (!createdir(META_TEMP)) {//U
+     DF_End();
      return;
    }
    _transit (name, OPENED);//A2
+   DF_End();
  }
```

Figure 6. DFix fast-forward patch for the bug in figure 2 (this patch, in green, is essentially the same with developers' patch; identities of A_1, A_2, and B are the inputs to DFix).

inside `DF_Start` and `DF_End` in figure 6) is used to support the above checking and rollback or fast-forward.

Algorithm 2: Patching a fault-timing bug[2].

```
   Input    : $A_1$, $A_2$, B
1  /*locate the start and end of buggy window */
2  $Loc_{Start}$ = Before ($A_1$);
3  $Loc_{End}$ = CFGExitNodes ($A_1$, $A_2$);
4  $\mathbb{OP}$ = CrashPersistentOperations ($A_1$, $A_2$);
5
6  /*Identify fastforward operations and their parameters*/
7  if (SafeFastFwd($A_1$, $A_2$)) then
8      $\mathbb{OP}_{FF}$ = RequireFastFwd ($\mathbb{OP}$);
9      $Params_{\mathbb{OP}_{FF}}$ = Slicing ($\mathbb{OP}_{FF}$, $A_1$);
10
11     /* Generate Fast-Forward Patch */
12     PATCH (
13       Log(START, $Params_{\mathbb{OP}_{FF}}$)  @ $Loc_{Start}$,
14       Log(END)                        @ $Loc_{End}$,
15       Log(OpID)                       @ After($\mathbb{OP}_{FF}$),
16       "IF(DF_CHECK()) {DF_FastFwd();}"
17                                       @ Before(B);
18 end
19
20 /*Identify rollback operations */
21 if (SafeRollback($A_1$, $A_2$)) then
22     $\mathbb{OP}_{RB}$ = RequireRollBack ($\mathbb{OP}$);
23
24     /* Generate Roll-Back Patch */
25     PATCH (
26       Log(START)                  @ $Loc_{Start}$,
27       Log(END)                    @ $Loc_{End}$,
28       Log(OpID, CheckPoint) @ Before($\mathbb{OP}_{RB}$),
29       "IF(DF_CHECK()) {DF_RollBack()};"
30                                   @ Before(B);
31 end
```

To generate these patches, DFix takes several steps as sketched out in Algorithm 2:

First (line 2–4 in Algorithm 2), DFix decides where to insert logging code to allow its patch checking whether and where node N_{fault} has crashed inside the bug-timing window, by analyzing every path connecting A_1 and A_2. In the example shown in figure 6, the locations of `DF_Start` and `DF_End` are decided at this step. The details are presented in Section 4.2.

Next, DFix checks whether a fast-forward patch is suitable for the bug (line 7 in Algorithm 2). If so, DFix identifies all the crash-persistent operations that potentially need fast-forward, denoted as $\mathbb{OP}_{FF}$ in Algorithm 2, decides how to pre-compute all the parameters needed by them before entering the buggy window (line 9), and generates the patch (line 12–17). The details of this step is presented in Section 4.3.

Finally, DFix checks whether a rollback patch is suitable for the bug (line 21). If so, it identifies all the crash-persistent operations that potentially need rollback, denoted as $\mathbb{OP}_{RB}$ in Algorithm 2, inserts a content-checkpoint function right before each of these operations (line 28), and generates the remainder of the patch similar to that of a fast-forward patch (line 26–30). The details are presented in Section 4.4.

Note that, unlike generic transactional rollback or fast-forward design, DFix puts much attention to patch simplicity — if too complicated, the patch will probably not be accepted in practice anyway. As a result, although DFix attempts to generate a rollback patch and a fast-forward patch for every fault-timing bug, either or both attempts may fail as simple rollback/fast-forward patches do not exist for every bug.

4.2 How to Observe the Buggy Timing

To allow a DFix patch checking whether and where node N_{fault} has crashed inside the bug window $A_1 - A_2$, logging code is inserted to mark the beginning and the end of the bug window, and all operations inside the window that might require rollback or fast-forward.

[2]Inputs are provided by the front-end detector and illustrated in Figure 3; `Patch X @ Y` indicates a DFix patch adding code `X` at location `Y`; the details of the logging function `Log` will be explained in Section 4.2 – 4.4.

Specifically, DFix statically analyzes every path connecting A_1 to A_2 on the control flow graph (CFG), and conducts the following checking for every operation I along the path.

First, if I has a successor I' that cannot reach A_2, DFix logs "END" right before I' to indicate the end of the bug-triggering fault window (line 14 and 27 in Algorithm 2) — that is how the two DF_End locations are decided in Figure 6.

Second, if I has side-effect beyond its local node N_{fault}, including global file-system updates[3] and message/RPC sending, DFix adds logging at I unless proved to be unnecessary later in Section 4.3 and 4.4. When I is a file operation, DFix places the logging right before and after the operation for rollback (line 28 in Algorithm 2) and fast-forward (line 15 in Algorithm 2) patches, respectively. This way, even if N_{fault} crashes between I and the logging, we can still guarantee correct patch semantics. Every log entry is appended with node and process ID, source file and line number. In figure 6, the two operations tagged with U and A_2 respectively are identified as having global side effects. Neither of them needs logging for different reasons: DFix decides U needs no fast-forward, which will be explained in Section 4.3, and hence does not insert logging for it; the logging for A_2 is combined with the end-of-buggy-window logging inside DF_End.

DFix currently only supports A_1 and A_2 coming from the same thread. We discuss how to obtain them for both subtypes of fault-timing bugs in Section 5.

4.3 Fast-Forward Design

At high level, a DFix fast-forward patch makes system updates that N_{fault} planned to but did not make due to its crash, which we refer to as *operation fast-forward*, so that the system can proceed as if N_{fault} crashed after the bug-triggering window.

Now that the DFix patch can decide which crash-persistent operations have yet executed (Section 4.2), we discuss below (1) how to identify operations that do not require fast-forward — the details behind the RequireFastFwd($\mathbb{OP}$) on line 8 of Algorithm 2; (2) how to get parameters of a to-be-fast-forward operation with its execution context wiped out by the crash — the details behind the Params$_{\mathbb{OP}_{\text{FF}}}$ on line 9 of Algorithm 2; (3) how to execute the operation *on behalf of* another node (i.e., N_{fault}) — the details behind the DF_FastFwd function in Algorithm 2 and Figure 6.

Note that, DFix identifies and gives up its fast-forward patching in either of these two cases, which is represented by the SafeFastFwd function in line 7 of Algorithm 2. First, there is a thread-waiting (e.g., a condition variable wait) between A_1 and A_2. In this case, fast-forwarding any operation after the wait may require fast-forwarding the waited-for thread too, which greatly complicates the patch. Second, whether a may-fast-forward operation will be executed and what are its parameter values cannot be decided *before* entering the buggy time window at runtime. We will elaborate on this case later.

Identify Operations That Need Fast-forward The crash-persistent operations that N_{fault} did not execute did not all require fast-forward. Specifically, DFix skips two types of operations I. First, I is post-dominated by another operation I' inside the buggy time window and they write to the same file, without any file read, synchronization, or communication operation in between. Second, a similar I' like the first case exists in code regions that post-dominate the failure site. For example, in figure 6, DFix static analysis identifies a deletedir operation that post-dominates the failure site and updates the same resource as createdir, which makes createdir unnecessary to fast forward[4]. Consequently, the DF_FastFwd function in figure 6 executes the other crash-persistent operation inside the buggy window — the _transit function tagged as A_2.

Fast-forward Parameter Preparation A DFix patch needs to pre-evaluate and record all the parameter values of all operations that may need to be fast-forwarded by another node before entering the buggy time window. To do so, DFix uses static slicing to identify all the instructions used to compute a parameter value, and clone these instructions to right before the buggy time window, just like how DFix pre-computes race-object identity in Section 3.5. Also like that in Section 3.5, DFix gives up its fast-forward patching if the pre-computation is not idempotent or relies on shared variables that might be updated by other threads during the buggy time window. In practice, file-update and message-sending parameters, like file paths, IP addresses, port numbers, and file/message content, are often constant or determined by thread-local variables, and hence can often pass the checking of DFix. At runtime, N_{fault} may crash right before or in between the pre-computation. since the pre-computation occurs right outside the buggy time window, such a node crash will not trigger any bugs.

Take the bug in figure 6 as an example. _transit has two parameters: name is a local variable and OPENED is a constant. In fact, name is updated to be this.region.toString() inside the process function. DFix identifies this assignment, and makes its patch pre-evaluates and records the content of this.region.toString() right before the buggy window, inside DF_Start. Here, region is a final field and hence its value is guaranteed not to change.

These pre-computed parameter values, as well as their type information, are logged to a file. DFix uses protocol buffer [7] for non-primitive parameters. When another node attempts fast-forward, it retrieves parameter information from the file and then conducts fast-forward. In figure 6, this is inside DF_FastFwd in the HMaster node.

[3]When B is executed and hence the failure is perceived during the restart of N_{fault}, local file updates are also identified.

[4]DFix has built-in knowledge about Java.io.File library.

Operation Fast-forward Finally, executing an operation on behalf of a different node is straight-forward for file updates, but requires API-specific handling for inter-node communication. The _transit function in figure 6 is a ZooKeeper API that any node can execute and achieve the same effect — certain znode is updated in ZooKeeper server with a particular value, which then causes ZooKeeper to send out corresponding messages. For socket message and RPC request sending, DFix pre-evaluates and records the destination IP address, port number, and message content in advance. We skip the detailed implementation for space constraints[5].

4.4 Rollback Design

At the high level, DFix's rollback patch follows the traditional transaction undo-log approach [30, 34]. When N_{fault} executes inside the buggy window, right before every crash-persistent write, the DFix patch records the original content and the update location. Later on, if N_{fault} crashes inside the buggy window, the DFix patch rolls back N_{fault}, pretending that N_{fault} crashed before the buggy window.

The main challenge is to balance patch simplicity and capability. At one extreme, to roll back arbitrary code, the patch could be very complicated in supporting coordinated rollback across many threads or nodes [19]. At the other extreme, existing single-node rollback techniques are simple, but too limited for many fault-timing bugs. For example, the rollback used in Section 3.4 to fix message-timing bugs and existing transactional memory techniques [34] do not support file or message rollbacks, and hence cannot help roll back crash-persistent operations. Existing file-system transaction techniques [6] do not handle message rollbacks; it is also impractical to change file-system in one bug patch.

Solutions DFix chooses a design that balances simplicity and capability: (1) it supports multi-thread and multi-node rollback, but is limited to the thread of A_1–A_2 on N_{fault} and the thread of the failure site where the crash is observed and rollback is launched, which we denote as B on N_B like HMaster-Thread 1 in figure 6; (2) it supports rollback of some common, but not all, heap accesses, file accesses, and messages.

Given a fault-timing bug, DFix first decides whether its rollback can fix this bug by statically analyzing every path between A_1 and A_2. DFix gives up its rollback patching once it identifies (1) a thread-signal, a thread-creation, or an event enqueue; or (2) an inter-node communication whose recipient handler cannot be statically identified; or (3) a recipient handler with side effects beyond the thread of B, A_1, and A_2. This is how the `SafeRollback` function on line 21 of Algorithm 2 works.

During the above static checking, DFix also identifies all operations that may require rollbacks — file updates and N_B's heap updates. For simplicity and performance concerns, DFix further prunes out operations that do not require rollback: (1) any write w dominated by write w' that updates the same object or file in the buggy window; (2) any file update dominated by the creation of its parent directory in the buggy window. This is how $\mathbb{OP}_{\text{RB}}$ is computed on line 22 of Algorithm 2.

After a bug passes the above checking and has all the may-rollback operations identified, DFix inserts content recording before every such operation (line 28 in Algorithm 2). For a heap update, an object clone to a shadow object is inserted. For a file update, the current prototype of DFix simply copies the whole to-be-updated file into a shadow file, as well as records the original file name. This simple scheme could lead to performance problems for large files, and our future work will improve this part by recording only the updated file content or leveraging copy-on-write mechanisms like `reflink` provided by some file systems.

One issue of this implementation is that a file update, if it is towards a global file, may already be read by another node before the rollback. This is usually not a problem, because if another node could read the inconsistent file image, another fault-timing bug would have been reported. Our future implementation can also try buffering global-file updates.

4.5 Patch Correctness Analysis

DFix relies on several assumptions to work correctly. First, the input provided by front-end bug detectors is correct. If this assumption does not hold, fast-forward or rollback may be conducted while node N_{fault} is still alive, which could lead to software misbehaviors. This bug-detection problem goes beyond the scope of DFix bug fixing. Second, the target software uses standard APIs for thread creation/join, signal/wait, and other synchronization operations. As discussed in Section 4.3 and 4.4, DFix may give up its patching when certain types of synchronization operations exist in the buggy time window. Automatically identifying arbitrary custom synchronization goes beyond the scope of DFix.

Under these assumptions, the DFix patch can correctly judge whether a fault-timing bug has occurred and make a best-effort fault handling like what developers' patches usually do. The logging design in Section 4.2 makes sure that the patch can correctly judge which operations require fast-forward or rollback, and the fast-forward and rollback design in Section 4.3 and 4.4 make sure that the patch does not introduce any new semantics and only makes the system behave as if the crash occurred later or earlier. There is also no risk of deadlocks, as the DFix patch does not conduct any blocking waits. DFix static analysis is conservative, and would give up chances for any optimization, like combining two updates to the same resource during fast-forward or rollback, if correctness cannot be guaranteed.

[5]Like previous tracing [54] and bug detectors [48, 49] for distributed systems, DFix requires built-in knowledge about common library interfaces.

5 Implementation Details & Limitations

DFix is implemented using WALA Java byte code analysis framework v1.3.5 [37] and JavaParser v3.2.4 for a total of around 5000 lines of code.

Bug Report Pre-processing Our current prototype uses bug reports automatically generated by a message-timing bug detector DCatch [48] and a fault-timing bug detector FCatch [49]. They are both dynamic bug detectors, and hence easily provide all the needed call stack and causality chain information. They also provide bug-triggering support, so that users can confirm whether a bug can truly cause severe failures before bug fixing.

The main information that is missing and has to be filled up by DFix is the buggy time window for fault order violations (figure 3c). FCatch reports *A* and *B* for this type of bug. *A* is clearly the end of the buggy fault time window. However, in practice, there is often a start point of the window — the bug is often not triggered if the node crashed too early. For example, the bug in figure 2 is a fault order violation, but the bug does not manifest if *RS* crashes earlier than `OpenRegion`. The current prototype of DFix uses the fault injection support provided by FCatch to figure out what is the start of the time window before *B* in the thread of *B*.

Library Specifications DFix identifies RPCs, message sending operations, event related functions, and synchronization following the related library interfaces, like `VersionProto` [9], `ProtoBase` [10], Hadoop `GenericEventHandler` [8], ZooKeeper [4], similar to previous work that dynamically analyzes and detects bugs in distributed systems [48, 49, 54]. DFix also contains built-in knowledge about Java basic file-operation APIs. A system that uses a different RPC, message, or event library would require customizing DFix with a new interface specification, which is easy to provide as also suggested by previous work [48, 49, 54].

Generate Source-code Patches DFix static analysis is conducted at WALA intermediate representation level, which WALA transfers from source code. At this level, all the object IDs are replaced by virtual register IDs. To generate source-code level patches for program developers to review, we translate the intermediate code back to source code leveraging a virtual-register-to-object map maintained by WALA.

Causality Clone Like previous tools that patch single-machine concurrency bugs [38, 39], DFix clones bug-related functions and applies its patch only in cloned functions, so that the patch only takes effect under the causality and calling context indicated by the bug report. Imagine the bug report indicates that a racing instruction is inside an RPC function `foo` remotely invoked by function `bar`. DFix would clone `foo` into a new function `foo_DCFix`, and add patching code in `foo_DCFix` but not `foo`. Similarly, DFix also clones `bar` into `bar_DCFix`, and changes `bar_DCFix` to invoke `foo_DCFix`. Since this clone technique is very similar to that in previous work [38], we skip the details due to space constraints.

Limitations of DFix DFix does not aim to fix all bugs. In fact, semantics-preserving simple patches do not exist for many bugs. As discussed earlier, DFix gives up on a message-timing bug, if the location that DFix chooses to initiate rollback, $\texttt{Loc}_{\text{CHECK}}$, ends up on a different node from *B*; DFix gives up on fixing a fault-timing bug, if (1) the bug-triggering fault window boundaries cannot be identified or cannot be located in one thread and (2) the rollback/fast-forward may involve more than one thread on the crashed node. DFix also gives up on using fast-forward to fix a fault-timing bug, if it cannot correctly pre-compute the context of the to-be-fast-forward operations; DFix gives up on using rollback to fix a fault-timing bug, if the rollback goes beyond the faulty node and the system-failure node. Furthermore, DFix fixes bugs that are triggered under specific call stack and causality stack reported by the front-end detector. DFix cannot fix bugs that are not reported.

6 Evaluation

6.1 Methodology

Benchmarks To evaluate DFix, we try *all* the 22 harmful bugs reported by front-end bug detectors in their papers [48, 49][6]. These include 10 message-timing bugs by DCatch [48], with 3 atomicity violations and 7 order violations, and 12 fault-timing bugs by FCatch [49], with 7 atomicity violations and 5 order violations. These are real-world bugs in 4 widely used systems: Cassandra key-value stores (CA), HBase key-value stores (HB), Hadoop Mapreduce (MR), and ZooKeeper metadata management service (ZK).

DFix successfully fixes 17 out of these 22 benchmark bugs. The remaining 5 include 4 fault-timing bugs whose bug-triggering time window A_1–A_2 goes beyond one thread and hence cannot be handled by DFix; and one message-timing bug that cannot be reliably triggered and hence is dropped from our benchmark suite.

Table 1 provides details for these 17 bugs. Clearly, developers took a long time to fix them[7], with only 4 of them fixed in < 1 week. Note that, these bugs took long time to fix **not** because they are considered not critical. Among them, 7 are tagged as top 10% important bugs by developers (i.e., 4 "blocker"s and 3 "critical"s), 7 are "major", 1 is "minor". The remaining 2 did not appear in bug-reporting systems.

[6]The original papers showed more, but some bugs are fixed once some other bugs are fixed and hence we did not double count.

[7]The time listed in Table 1 includes both bug-understanding time and patch-design time. We can only see that both are very time consuming, but cannot tell exactly how much time went to each from bug reports.

Table 1. DFix benchmarks used in Section 6.2 – 6.3. Those 4/5-digit numbers are bug-IDs in bug databases.

ID	App-BugID	LoC	Manual Fixing	Description
Root Cause: Message Atomicity Violations				
AM1	CA-1011	61K	114 days	Another node joins the cluster while one node is bootstrapping.
AM2	HB-4539	188K	5 days	A ZKnode delete happening between a creation and a delete causes double delete exception.
AM3	HB-4729	213K	27 days	A ZKnode create happening between a creation and a delete causes double create exception.
Root Cause: Message Order Violations				
OM1	CA-extra1	61K	182 days	The seed node sends the message so late that the follower shuts down itself.
OM2	MR-3274	1.2M	4 days	A map entry is removed before read by rpc handler from another node.
OM3	MR-4637	1.3M	48 days	An update event arrives in statemachine before it transferring to ASSIGNED state.
OM4	ZK-1144	102K	8 days	A map entry is read by a socket message handler before local node puts it to the map.
OM5	ZK-1270	110K	7 days	A list element is read by a socket message handler before local node puts it to the list.
OM6	ZK-1194	110K	45 days	The follower registers the epoch before the leader.
Root Cause: Fault Atomicity Violations				
AF1	HB-2611	137K	998 days	While RS taking over HLog, crashing between ZKHLog create and delete causes data loss.
AF1	HB-3596	137K	12 days	While RS taking over HLog, crashing between ZKLock create and delete causes data loss.
AF2	HB-12241	137K	918 days	While RS taking over HLog, crashing between ZKpeerID create and delete causes data loss.
AF3	ZK-1653	67K	272 days	A node crashes between two file updates cause data inconsistency
Root Cause: Fault Order Violations				
OF1	CA-5393	159K	22 days	While taking snapshot, the leader will hang if the follower loses its ack message.
OF2	CA-6415	159K	5 days	While collecting merkle tree, the leader will hang if the follower loses its tree message.
OF3	CA-extra2	159K	91 days	While processing antientropy, the leader will hang if the follower loses its response.
OF4	HB-10090	536K	5 days	HMaster will hang if the META-region open message is lost.

Setup We use two machines, with Intel i7-3770 CPU, 8GB RAM, Ubuntu 14.04, and JVM v1.7. We evaluate whether DFix can fix a bug, as well as the performance and simplicity of the resulting patches. We also compare DFix with manual patches and alternative naïve patching strategies.

6.2 Overall Result

Functionality As shown in table 2, DFix successfully fixes all 17 bugs listed in table 1; naïve patch strategies like adding locks around racing operations and restarting the crashed node after local-file cleanups can only fix 2 benchmarks, OM1 and AF4 (they also cannot fix any of the 5 not in table 1 and 2 that DFix cannot fix); there is one benchmark, AF2, that even the final manual patch did not completely fix — the manual patch only added sleep to lower the failure probability.

We make our best effort in evaluating patch correctness through stress testing, code inspection, and comparison with manual patches. Our stress testing inserts random-length sleeps around racing operations for message-timing bugs, and injects node faults randomly in the bug-triggering time window. Under this setting, the original software fails about 50 – 100% of times in 20 runs; the software with DFix patches never fail in our experiments.

Naïve patches for message-timing bugs apply traditional locks and condition-variable signals/waits around racing operations. It fails to fix 8 out of 9 message-timing bugs. It does not apply to AM2 and AM3, as racing objects are znodes on third-party ZooKeeper servers and we cannot change the ZooKeeper library. It causes deadlocks for 6 benchmarks, due to circular waits on event handling thread (OM3), message handling threads (AM1, OM2), and locks (OM4, OM5, OM6).

Table 2. Overall results. (*: manual patching takes so long that the software has changed too much for performance comparison; $T_{correct}$: no sleep inserted, baseline is original software; T_{buggy}: sleep inserted to trigger the bug, baseline is manual patch; #: goes down with shorter sleep.)

	Is the bug fixed?			Overhead (%)		
				$T_{correct}$		T_{buggy}
ID	Manual	DFix	Naive	Manual	DFix	DFix
AM1	✓	✓	$\times_{hang}$	*	0.5	*
AM2	✓	✓	×	0.7	0.2	-0.1
AM3	✓	✓	×	2.3	2.4	-1.6
OM1	✓	✓	✓	*	0.4	*
OM2	✓	✓	$\times_{hang}$	0.2	0.4	21.2#
OM3	✓	✓	$\times_{hang}$	0.2	0.7	23.3#
OM4	✓	✓	$\times_{hang}$	-1.2	0	0.3
OM5	✓	✓	$\times_{hang}$	-2.1	0	0.2
OM6	✓	✓	$\times_{hang}$	-0.5	0.8	-0.1
AF1	✓	✓	×	*	-4.2	*
AF2	✓✗	✓	×	13.0	0.1	-0.4
AF3	✓	✓	×	*	1.6	*
AF4	✓	✓	✓	-3.4	-4.0	-0.1
OF1	✓	✓	×	0.2	1.6	-1.1
OF2	✓	✓	×	-2.6	0.6	2.1
OF3	✓	✓	×	*	2.4	*
OF4	✓	✓	×	3.3	-4.3	3.4

Naïve patches for fault-timing bugs always restart the crashed node after clean up local temporary files (e.g., we clean up the hadoop.tmp.dir directory). This strategy can fix 1 out of 8 fault-timing bugs (i.e., AF4). For the 4 order violations, simply restarting the crashed node does not help sending a message that another node is waiting for; for 3 atomicity violations (AF1, AF2, AF3), the node crash left inconsistency among global files and local cleanup does not work. Furthermore, restart without local cleanup cannot fix any bugs.

Performance We have measured performance of DFix patches under both correct timing and bug-triggering timing, indicated by $T_{correct}$ and T_{buggy} in table 2.

DFix patches introduce almost no overhead under correct timing, comparing with the original software with no code changes ($T_{correct}$ in table). **No** sleep/fault was injected and the baseline run-time varies from 1 second to half a minute in different benchmarks. We take the average of 40 runs for every performance number in the table. As we can see, the overhead of both manual and DFix patches are negligible. The only exception is that, the manual patch inserts a random sleep for AF2, which can lead to huge overhead non-deterministically and leads to a 13% overhead on average in our experiments. Both manual patches and DFix patches occasionally incur negative overheads. The amount is negligible and is caused by fluctuation of the running environment.

DFix patches are as fast as manual patches under the bug-triggering timing for all but 2 benchmarks, OM2 and OM3 (T_{buggy} in table). We use the manually patched software as baseline, because the original software fails under bug-triggering timing. For OM2, OM3, manual patches actually changed the original software data structure, related processing algorithms and semantics, so that the B operation no longer needs to wait for A operation. The 20% overhead of DFix in actually *inevitable* for any semantic-preserving patches — to trigger the bug, 1–20 seconds of random sleeps are inserted, which not only delays A but also B to guarantee B executes after A. In manual patch, B does not wait for A and hence is not delayed for that 1–20 seconds. In fact, this overhead is much smaller when we trigger bugs with short sleeps.

There are 3 benchmarks, AF1, AF3, and AF4, that DFix can generate two patches, a fast-forward and a rollback patch. Table 1 only reports the performance of their fast-forward patches. The performance of rollback patches is similar.

6.3 Patch details

Message-timing patches Among the 9 message-timing bugs, 5 of them (AM1, AM2, AM3, OM2, and OM3) require inter-node rollbacks, while the other 4 are fixed through intra-node rollbacks. Among these 4, three of them have their race operations inside synchronized blocks. Consequently, DFix identifies rollback destination outside the synchronized blocks to avoid deadlocks. Object-specific flag IDs (Section 3.5) are crucial for 5 of them (AM1, AM2, AM3, OM2, OM3).

As discussed in Section 3.6, DFix proves 7 of 9 bug patches to be deadlock free – the waited-for instruction is guaranteed to execute. For the remaining two benchmarks (AM1 and OM2), their waited-for instructions are inside a branch body of the heartbeat handler. Static analysis cannot guarantee those instructions will execute and also cannot identify a location, after which those instructions definitely will not execute due to the periodic nature of heartbeat protocol. DFix uses timeouts in all its patches to prevent infinite rollbacks.

Table 3. DFix fault-timing patches.

	Fast-forward	Rollback	#crash-persistent Ops	
ID	Fixed?	Fixed?	raw	optimized
AF1	✓	✓	1	1
AF2	×	✓	4	2
AF3	✓	✓	1	1
AF4	✓	✓	2	2
OF1	✓	×	1	1
OF2	✓	×	1	1
OF3	×	✓	1	1
OF4	✓	×	18	2

Fault-timing Patches As shown in table 3, fast-forward and rollback each can fix most but not all of these 8 benchmarks. Fast-forward can fix all but two bugs. For AF2 and OF3, the parameter pre-evaluation depends on shared variables whose content might be changed later during N_{fault}'s execution in the buggy window. Consequently, DFix gives up. Rollback can fix all but three bugs, OF1, OF2, and OF4. The buggy windows of these three bugs contain inter-node communication that DFix cannot handle. These two strategies nicely complement each other and work together to help DFix fixes all these 8 bugs.

The last two columns of table 3 present the number of crash-persistent operations inside each buggy time window. One bug's window only involves file updates; 3 only involve message sending; the remaining 4 involve both types of operations. For AF2 and OF4, some of their crash-persistent operations are considered as unnecessary for rollback/fast-forward through DFix static analysis, as discussed in Section 4.3 and 4.4.

Simplicity DFix patches for all the 17 benchmarks range between 5 to a little over 20 lines of code, excluding utility functions like logging and wait-with-timeouts in DFix library. The path-size differences are mainly affected by the complexity of pre-computing the race-object identity, and the number of crash-persistent operations.

Patch Generation The static analysis used by DFix to generate patches is scalable, as the analysis is applied only to bug related call stacks and causality stacks. All the patches are generated within 200 seconds in our experiments.

Comparison with Manual Patches 10 out of 17 bugs are fixed by developers using *exactly the same* strategy as DFix. One bug's manual patch (AF2) does not completely work. For the other 6, developers redesigned algorithms/data-structures, so that the race operations either disappear (3 bugs) or the originally unexpected timing no longer causes failures (3 bugs). The former three patches are much more complicated, and the latter three are simpler than DFix patches.

Here are two examples where patches manually developed by programmers use exactly the same strategy as patches automatically generated by DFix. Bug ZK-1270 (OM5) is a message-timing bug: O_1, a local operation, races with O_2, an operation inside a message-handler. Manual patch introduces a variable to indicate if O_1 has executed (like DFix in Section-3.3) and rolls back O_2's message handler when the checking fails (like DFix in Section-3.4). Manual-patch's re-execution region is the same as DFix-patch. Bug ZK-1653 (AF4) is a fault-timing bug: a node writes new epoch-ID to file A and then B. If it crashes in between, restart would fail due to inconsistent epoch-IDs in A-and-B. The manual patch creates a flag-file to indicate whether A and/or B was updated, just like DFix. Using that file, restart checks if the crash was in between and, if so, copies new epoch-ID from A to B, just like how DFix fast-forwards B's missing update.

7 Related Work

Automated Bug Fixing Motivated by the huge cost in bug fixing and its huge impact to software availability, much research has been conducted for automating patch generation recently [22, 41, 65, 69]. In addition to single-node concurrency bug fixing techniques discussed in Section 1.1, other auto-fixing techniques have also been proposed. Some of them focus on specific types of bugs that are unrelated to distributed timing bugs [27, 60]; some use program verification and synthesis techniques to find patches that fit a specific template (e.g., only change one variable or one operator in software) [18, 52]; and many techniques use heuristics to search software mutation space to find mutations that can pass all regression tests [44]. None of them suits the problem of fixing distributed timing bugs, where the program space is huge and the bug exists in timing, instead of computation logic.

Distributed Timing-Bug Detection Bug detection [17, 48, 49] and model checking tools [32, 43, 45, 47, 63] have been proposed to detect distributed timing bugs. They are all potential front ends to auto-fixing tools like DFix.

Note that, bug detection tools can help but can**not** replace bug fixing tools. As we can see in the design of DFix, even after the details of a bug are known, there are still many program analysis and reasoning required to produce a patch. It is desirable to relieve developers from such costly and error-prone effort.

Improving Distributed System Availability Program verification and auto-proving [35, 68] is a promising direction to build highly available distributed systems. Existing techniques cannot scale to large distributed systems with many protocols, and sacrifice performance greatly.

PAR [13] uses protocol-specific knowledge to design correct disk-failure recovery routines in cloud storage that is built upon replicated state machine (RSM) protocol. It does not help fixing our bugs, as none of them is from RSM protocol. Olive[61] describes an approach, lock with intent, that provides exactly-once semantics for a transaction of operations that work on certain type of global storage. It is very effective for fault tolerance of certain type of transactions, and can help fix some of our fault-timing bugs like AF1 – AF3, but cannot help other fault-timing bugs that go beyond storage problems and involve more complicated storage systems. These fault tolerance techniques do not help fix the message-timing bugs.

8 Conclusion

Distributed timing bugs widely exist in distributed systems and take long time to fix. Fixing them encounters unique synchronization and scope challenges. In this paper, DFix explores using carefully designed rollback and fast-forward to handle bug-triggering timing and fixes distributed timing bugs without introducing new bugs. DFix is just a starting point. Future work can further explore the design space of patch coverage and patch simplicity.

Acknowledgments

We would like to thank Shaz Qadeer, our shepherd, and the anonymous reviewers for their insightful feedback and comments. This research is supported by NSF (grants CCF-1837120, CNS-1764039, 1563956, 1514256, IIS-154654) and CERES Centerfor Unstoppable Computing.

References

[1] 2013. ZooKeeper 1653. https://issues.apache.org/jira/browse/ZOOKEEPER-1653. (2013). Accessed: 2013-11-26.
[2] 2017. HBase 10090. https://issues.apache.org/jira/browse/HBASE-10090. (2017). Accessed: 2017-09-16.
[3] 2017. MapReduce 4637. https://issues.apache.org/jira/browse/MAPREDUCE-4637. (2017). Accessed: 2017-09-16.
[4] 2017. ZooKeeper. https://zookeeper.apache.org/. (2017). Accessed: 2017-09-16.
[5] 2017. ZooKeeper 1270. https://issues.apache.org/jira/browse/ZOOKEEPER-1270. (2017). Accessed: 2017-09-16.
[6] 2018. Btrfs Rollback. https://ramsdenj.com/2016/04/05/using-btrfs-for-easy-backup-and-rollback.html. (2018). Accessed: 2018-04-30.
[7] 2018. Google Protocol Buffer. https://developers.google.com/protocol-buffers/docs/reference/overview. (2018). Accessed: 2018-05-01.
[8] 2018. Hadoop AsyncDispatcher. https://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/yarn/event/AsyncDispatcher.html. (2018). Accessed: 2018-05-01.
[9] 2018. Hadoop VersionProto. https://blog.woopi.org/wordpress/files/hadoop-2.6.0-javadoc/org/apache/hadoop/yarn/proto/

YarnServerCommonProtos.VersionProto.html. (2018). Accessed: 2018-05-01.

[10] 2018. HBase ProtoBase. http://www.grepcode.com/file/repository.cloudera.com/content/repositories/releases/org.apache.hadoop/hadoop-yarn-common/2.3.0-cdh5.1.4/org/apache/hadoop/yarn/api/records/impl/pb/ProtoBase.java?av=h. (2018). Accessed: 2018-05-01.

[11] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing event race errors by controlling nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 289–299.

[12] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[13] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Protocol-Aware Recovery for Consensus-Based Storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 15–32. https://www.usenix.org/conference/fast18/presentation/alagappan

[14] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated Crash Vulnerabilities. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 151–167. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/alagappan

[15] Matt Bishop and Michael Dilger. 1996. Checking for Race Conditions in File Accesses. *Computing Systems* 2, 2 (1996), 131–152. http://www.usenix.org/publications/compsystems/1996/spr_bishop.pdf

[16] Nikita Borisov and Robert Johnson. 2005. Fixing Races for Fun and Profit: How to Abuse atime. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. https://www.usenix.org/conference/14th-usenix-security-symposium/fixing-races-fun-and-profit-how-abuse-atime

[17] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. 2017. Serializability for eventual consistency: criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 458–472.

[18] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic debugging. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 121–130.

[19] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[21] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services.. In *OSDI*. 217–231.

[22] Eduardo Faria de Souza, Claire Le Goues, and Celso Gonçalves Camilo-Junior. 2018. A Novel Fitness Function for Automated Program Repair Based on Source Code Checkpoints. (2018).

[23] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[25] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices* 48, 6 (2013), 321–332.

[26] Dennis Michael Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. Replay debugging for distributed applications. (2006).

[27] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. 2017. Automatically Repairing Network Control Planes Using an Abstract Representation. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 359–373. DOI: http://dx.doi.org/10.1145/3132747.3132753

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *ACM SIGOPS operating systems review*, Vol. 37. ACM, 29–43.

[29] Patrice Godefroid and Nachiappan Nagappani. 2008. *Concurrency at Microsoft – An Exploratory Survey*. Technical Report. MSR-TR-2008-75, Microsoft Research.

[30] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. http://portal.acm.org/citation.cfm?id=573304

[31] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*.

[32] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 265–278.

[33] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. 2013. Failure Recovery: When the Cure Is Worse Than the Disease. In *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*. https://www.usenix.org/conference/hotos13/session/guo

[34] Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory, 2Nd Edition* (2nd ed.). Morgan and Claypool Publishers.

[35] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 1–17.

[36] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: recording local executions to reproduce concurrency failures. In *PLDI*.

[37] IBM. 2017. Main Page - WalaWiki. http://wala.sourceforge.net/wiki/index.php/Main_Page. (2017).

[38] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 389–400.

[39] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated Concurrency-Bug Fixing.. In *OSDI*, Vol. 12. 221–236.

[40] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. 2008. Deadlock Immunity: Enabling Systems To Defend Against Deadlocks. In *OSDI*.

[41] René Just, Chris Parnin, Ian Drosos, and Michael D Ernst. 2018. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 287–297.

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

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

[44] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair:

Fixing 55 out of 105 Bugs for $8 Each. In *ICSE*.

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

[46] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. 2016. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 517–530.

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

[48] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *ASPLOS*.

[49] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 419–431.

[50] Peng Liu, Omer Tripp, and Charles Zhang. 2014. Grail: Context-aware Fixing of Concurrency Bugs. In *FSE*.

[51] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. 2007. WiDS checker: Combating bugs in distributed systems. (2007).

[52] Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 133–146.

[53] Brandon Lucia and Luis Ceze. 2013. Cooperative empirical failure avoidance for multithreaded programs. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*. 39–50. DOI: http://dx.doi.org/10.1145/2451116.2451121

[54] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 378–393.

[55] IHS Markit. 2016. Businesses Losing $700 Billion a Year to IT Downtime, Says IHS. http://news.ihsmarkit.com/press-release/technology/businesses-losing-700-billion-year-it-downtime-says-ihs. (2016).

[56] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

[57] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen S Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat Datacenter Storage.. In *OSDI*. 1–15.

[58] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, and others. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 87–102.

[59] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: treating bugs as allergies—a safe method to survive software failures. In *Acm sigops operating systems review*, Vol. 39. ACM, 235–248.

[60] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. 2012. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 277–287.

[61] Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. 2016. Realizing the Fault-Tolerance Promise of Cloud Storage Using Locks with Intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 501–516. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/setty

[62] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D Keromytis. 2009. Assure: automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News* 37, 1 (2009), 37–48.

[63] Jiri Simsa, Randy Bryant, and Garth A Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems.. In *SSV*.

[64] Ian Sommerville. 2006. *Software Engineering: (Update) (8th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[65] Akito Tanikado, Haruki Yokoyama, Masahiro Yamamoto, Soichi Sumi, Yoshiki Higo, and Shinji Kusumoto. 2017. New Strategies for Selecting Reuse Candidates on Automated Program Repair. In *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, Vol. 2. IEEE, 266–267.

[66] Theregister. 2017. AWS's S3 outage was so bad Amazon couldn't get into its own dashboard to warn the world. https://www.theregister.co.uk/2017/03/01/aws_s3_outage/. (2017).

[67] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlk. 2008. Gadara: dynamic deadlock avoidance for multthreaded programs. In *OSDI*.

[68] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 357–368.

[69] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2017. Automated Bug Removal for Software-Defined Networks.. In *NSDI*. 719–733.

[70] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *OSDI*.

[71] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 19–33.