DFIX: AUTOMATE DISTRIBUTED CONCURRENCY BUG FIXING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER

DEPARTMENT OF COMPUTER SCIENCE

BY
GUANGPU LI

CHICAGO, ILLINOIS
DECEMBER 2017
# TABLE OF CONTENTS

LIST OF FIGURES ........................................ iv
LIST OF TABLES .......................................... v
ACKNOWLEDGMENTS ...................................... vi
ABSTRACT ................................................ vii

1 INTRODUCTION ........................................ 1
   1.0.1 Motivation ....................................... 1
   1.0.2 Contributions .................................... 3

2 BACKGROUND .......................................... 5
   2.0.1 DCbugs .......................................... 5
   2.0.2 Timing relationship in distributed systems ...... 5

3 DESIGN ............................................... 9
   3.1 DFix Overview ...................................... 9
   3.1.1 Challenges ...................................... 9
   3.1.2 Our solutions .................................. 10
   3.2 DFix Bug Fixing .................................... 13
      3.2.1 RestartSite Selection ......................... 13
      3.2.2 CheckSite Selection ........................... 15
      3.2.3 Rollback ...................................... 17
      3.2.4 Creating condition flags ...................... 19
      3.2.5 Deadlock Analysis ............................ 24
      3.2.6 Others ........................................ 27

4 EVALUATION ......................................... 29
   4.0.1 Methodology .................................... 29
   4.0.2 Overall Result .................................. 30
   4.0.3 Understanding Result ........................... 31
   4.0.4 Patch details ................................... 31
   4.0.5 Performance .................................... 32

5 RELATED WORK ...................................... 34

6 CONCLUSION ......................................... 36

REFERENCES .......................................... 37
LIST OF FIGURES

1.1 Example ............................................................. 2

2.1 Message Cancel Task1 will remove an entry in a map in AM, which will be read
in message Read Task1. If the read result is null, The Task1 container will hang. 6

2.2 This is a bug in HBase. HR1 and HR2 both create and delete zknode with the
same path through message. The create message will be ignored if the zknode
already exists. If (create, delete) region is interleaved by another create message,
there will be a double delete exception. ........................................ 6

3.1 This is the overview of different patches for order violation. (a) is the patch based
on synchronization primitive. Wait will delay the operation until A is executed.
(b) is the failure recovery. It will keep rollback until the failure does not happen.
(c) is the patch from DFix. If will keep rollback and predicting if the bug will
happen at check site until F is set after A. ........................................ 11

3.2 This is the overview of different patches for atomicity violation. (a) is the patch
that protect the atomicity by lock. (b) is the failure recovery. It will keep rollback
until the failure does not happen. (c) is the patch from DFix. It protects the
atomicity region by a flag F. And if CheckSite find F is set, it will rollback. . . 12

3.3 RestartSite search. DFix initializes the RestartSite location at the delayed oper-
ation and begins backtrack along the causality chain. First, DFix moves the
RestartSite out of the synchronized region. Then, DFix moves the RestartSite
to the event enqueue operation because the current RestartSite is inside a event
handler. In the end, DFix moves RestartSite the message sending operation, as
it finds the RestartSite is in the message handler. .......................... 13

3.4 Example about rollback from inside an event handler to rpc callsite int thread
T1. After event enqueue operation put(e1) in thread T2, there is a timed wait
to make it synchronized. f.wait() wait f is set somewhere. f.set(v) sets a specific
value v to f. If timeouts, it will throw an exception to rpc callsite. And the wait
is signaled after check passes in CheckSite in thread T3. To avoid processing
multiple same e1, DFix adds a check-drop at the beginning of its handler . . . 20

3.5 the rectangles are instance of A. Circles are instances of B. White operations
access the same memory location. Three grays access another one. Bring order
across white and gray is helpless for fixing. Associating the ordered operations
with address provides better order control. ..................................... 21

3.6 Domination and post-domination analysis. For operation F, A, B, C dominate it.
For operation C, it is post-dominated by F. A is an event enqueue operation. B is
the beginning of event handler. So A always dominates B. And B post-domimates
A. ................................................................. 26

3.7 If only clone once, it is hard to different e1 is a repeat event or a new event from
another invocation. DFix clones it twice with another path that contains ’’. All
the repeat events happen in the second path. In this way the new event will not
be dropped mistakenly. .......................................................... 28
LIST OF TABLES

4.1 DFix Benchmarks. ................................................................. 29
4.2 DFix Overall Result. M-DL means the deadlock caused by message block. E-DL means the deadlock caused by event block. L-DL is the deadlock caused by lock. X means no naive solution ................................................................. 30
4.3 DFix Bug Understanding. .......................................................... 31
4.4 DFix Patch Details. ................................................................. 32
4.5 DFix Patch Performance. .......................................................... 33
ACKNOWLEDGMENTS

Fist and foremost, I would like to thank my advisor Prof. Shan Lu for supervising me in these two years. She taught me the action is more important than words by her action. She also taught me the vision is more important than action by her vision. She is so kind that she can bear my mistakes that puts our group in danger sometimes. I feel fortunate to listen her detailed suggestions about communication, time management, substantial research.

I want to thank Haopeng Liu, Yuxi Chen, Shu wang, Chi Li, Junwen Yang, Totem Tang, Mingzhe Hao, Huan Ke, Huaicheng Li and Ding Yi for their help during my two years in University of Chicago.

Finally, I want to thank my parents Shuicai Li and Jingying Zhao for their love and support.
Fixing bugs in distributed system is challenging and time-consuming. Fixing concurrency bugs in distributed system is especially difficult as many nondeterminism involved. The various synchronization mechanisms inside message handling and event-driven processing make the timing hard to control. Existing tools that focus on single machine software fixing cannot handle these synchronization mechanisms, which introduces deadlock in patch.

We present DFix, a tool that automates the distributed system concurrency bugs fixing process without introducing new functionality, new components, and huge performance loss. DFix proposes a check-rollback solution for fixing. DFix starts with the bug reports from exiting distributed concurrency bug detection tool. Through static analysis, DFix chooses the suitable location to check if need rollback to delay an operation. The check condition is related to the address of delayed operation, which provides fine granularity timing control. Our evaluation shows that DFix can fix seven real-world bugs automatically within less than one percent overhead.
CHAPTER 1
INTRODUCTION

1.0.1 Motivation

Many applications in big data field leverage cloud systems such as scale-out storage systems[8, 10, 14, 33], synchronization services[18, 4], computing frameworks [9, 32] and cluster management services[17, 37] as their backbones. While cloud systems bring scalability and performance benefits, the reliability of these systems cannot always reach the expectation of users. Particularly, the reliability of these systems is often threatened by distributed concurrency bugs — software bugs triggered by special timing among distributed operations. These bugs widely exist in distributed systems [26], causing severe failures and are difficult to discover and diagnose due to their non-determinism natures.

Recent works [11, 27, 36, 25, 22, 15, 29] try to discover nondeterminism problem of distributed system through concurrency bug detection.

Unfortunately, system reliability does not improve until after these bugs are fixed, yet fixing distributed concurrency bugs, short for DCBugs, is often time consuming and error prone. A previous study [3] of open-source distributed system shows that it takes 82 days on average to fix a DCBug.

Figure 1.1 illustrates an real example of DCbug from MapReduce [1]. The bug is triggered by unexpected timing between NodeManager(NM) and ResourceManager(RM). After user submit a job, on the one hand, RM would check running health of one task from ApplicationManager(AM). This check is implemented as an RPC function call. Inside the RPC handler, it will put a health-check event to an FIFO event queue associated to the checked task. On the other hand, NM would notice AM after a task is launched. The notification is also implemented as another RPC function call which puts a launch-notification event to the queue. If health-check event is handled before launch-notification event, the job will abort because someone wants to check the running health of an un-running task. Though the
bug is caused by the order where health-check event enters the queue early than launcher-notification events, to fix this bug automatically is no trivial.

There are challenges unique for distributed systems that cause traditional synchronization primitives unsuitable for fixing timing problems in distributed systems.

- Distributed. Bug-related code regions may be on different nodes and consequently some traditional synchronization primitives like locks, signals, and waits do not work.

- Deadlock prone. Blocking wait inside event handlers or RPC handlers could all cause deadlocks, as an event/RPC handler implicitly holds resources (i.e., the limited number of event-handling or RPC-handling threads) that might be needed by another code region to execute the unblocking operation.

- Timeout prone. Delays, which are necessarily to get around a buggy timing, may conflict with existing fault tolerance mechanisms and cause cause time-out exceptions, which can then lead to unexpected exceptions or even message losses.
• Complexity. The mix of asynchronous and synchronous computation (e.g., event processing vs. thread parallelism in every node) and communication (e.g., blocking RPC calls vs. non-blocking socket messages among different nodes), the long and cross-node causality chains, the fault tolerance concerns makes the reasoning about program semantics and timing relationships complicated.

To attack concurrency bug fixing, there are a body of automated concurrency bug fixing tools[20, 19, 2]. But they only work well for signal process multi-threading software. [19, 20] introduce the synchronization primitives to eliminate the buggy timing. But their analysis treat event or message handler as normal thread and insert the synchronization primitives. This will cause deadlock and timeout. [2] proposes a controller for event process to manipulate the timing between event processing. However the manipulation policies are application specific. Moreover, it does not handle the timing related to message neither.

As these unique challenges in distributed system, developers are in great need of automatic technology in fixing DCbugs.

this paper proposes DFix : a tool that fixes DCBugs automatically without introducing new functionality problem, new synchronization service and significant performance degradation.

1.0.2 Contributions

DFix automatically fixes DCBugs reported by DCbug-detection tools through thorough static and dynamic program analysis.

At high-level, different from existing tools that fix concurrency bugs through either proactive blocking waits (i.e., adding locks) or reactive failure recoveries, DFix fixes DCbugs through error-prediction and rollback-re-execution. This scheme avoids the deadlock-prone and timeout-prone limitations associated with blocking waits, and also provides flexibility to ease distributed rollback and re-execution, which we will present in details in Section 3.1.
For example, for the illustration example shown in Figure 1.1, DFix -patch could predict whether a buggy interleaving is going to happen at xxx and rollback to xxx when the checking fails. This scheme successfully avoids the deadlock problem discussed earlier and guarantees not to introduce new problems into the system.

Of course, there are many detailed challenges in implementing the above fix strategy — how and where to conduct error prediction? where should the re-execution starts? how to correctly rollback and re-execute? DFix carefully handles all these problems while considering the complicated and diverse concurrency and communication schemes in distributed systems. We will present these details in Section 3.2.

We have evaluated DFix on seven real-world DCbugs in Cassandra, HBase, MapReduce, and ZooKeeper. DFix successfully fixes all these bugs without introducing new bugs or unnecessary performance degradation.
CHAPTER 2
BACKGROUND

2.0.1 DCbugs

DCbugs are triggered by incorrect timing of inter-node communication in distributed systems. The incorrect timing leads to unexpected and conflicting accesses to shared memory objects in a node, referred to as the root-cause node. The resulting errors sometimes propagate beyond the root-cause node [26].

For example, in Figure 2.1, the bug is triggered that the Read Task1 message arrives later than cancel message. The data race happens in AM. But the actual symptom is a hang in Task1 Container.

DCBug root causes Overall, concurrency bug consists two main categories: order violation(OV) and atomicity violation(AV).

Order violations are about two operations $A$ and $B$. Software behaves correctly if $A$ executes before $B$, and incorrectly if $A$ executes after $B$. For example, in Figure 2.1, the bug is caused by removing the entry executes before the read operation.

Atomicity violations occur when a code region, referred to as atomicity region $R$, is unserializably interleaved by an operation $r$. For example, in Figure 2.2, The atomicity region is (create, delete) of a zknode. And if this region is interleaved by another create of the same zknode, there will be a double delete exception.

2.0.2 Timing relationship in distributed systems

Fixing concurrency bugs is fundamentally about enforcing timing relationships and reasoning about happens-before relationships [24]. We briefly discuss several timing-related primitives or operations that are unique to distributed systems, and the general happens-before model in distributed systems [29].
Figure 2.1: Message Cancel Task1 will remove an entry in a map in AM, which will be read in message Read Task1. If the read result is null, The Task1 container will hang.

Figure 2.2: This is a bug in HBase. HR1 and HR2 both create and delete zknode with the same path through message. The create message will be ignored if the zknode already exists. If (create, delete) region is interleaved by another create message, there will be a double delete exception.
Intra-node timing There are two types of threads in distributed systems: regular threads and asynchronous event-handling threads.

Timing among operations in regular threads is determined by programming order, which have been well studied for decades in single-machine concurrency bug detection and fixing work [19, 20, 35, 31, 13].

Timing relationship that involves asynchronous threads has been studied in recent years. Each event-handling thread serves an event queue. Any thread can enqueue an event into the queue and continue its execution without waiting for the event. The event will later be dispatched to a handling thread which executes a corresponding event handler.

Note that, waiting inside the event-handling thread can potentially lead to deadlocks. For example, Figure 1.1 depicts a situation where the condition-variable wait operation blocks the event-handling thread, which then causes the signal operation never being able to execute (i.e., a deadlock).

Inter-node timing Across different nodes, distributed systems have two types of message communication models. One is asynchronous socket message. And the other is RPC message. The only difference is that RPC sending site waits for the result of message processing before continue but socket not.

Note that, distributes systems always use a timer for blocking communication (e.g., RPCs) between nodes for fault tolerance. If the communication takes for too long (e.g., when the RPC function is waiting for a resource for too long), a time out would fire, which can then cause a node to be mistakenly judged as unavailable and cause severe job and system failures. For example, the time-out setting for heartbeat is 10ms.

Distributed happens-before analysis Concurrency bug detection always contains happens-before analysis. Recent DCbug detection tool [29] also builds a happens-before model for distributed systems and conducts dynamic happens-before analysis during its bug detection.

In this paper, we define the causality chain to refer a trace that reach to one specific
executed instruction considering the message processing, event processing, and programming order. Causality chain is an important clue for tracing the reason of occurrence of one operation in distributed system. Causality chain is a linked list that records the causality of one operation. Every element in the list is an instruction. The previous instruction is the reason of occurrence of next one. Usually in distributed system, three reasons are considered: (1) function caller is the reason of callee, (2) event enqueue operation is the reason of its handler, and (3) message sending is the reason of its handler.
CHAPTER 3
DESIGN

3.1 DFix Overview

3.1.1 Challenges

There are traditionally two approaches to fixing timing problems in multi-threaded software: (1) proactive synchronization using locks, condition variable signals and waits; (2) reactive failure recovery using checkpoint and rollback. Neither approach is suitable for fixing timing problems in distributed systems.

Strawman-1: locks and condition variables  An inherent property of locks and condition variables is that they could both block a thread to wait for locks or signals. This property could lead to fatal time-out exceptions and/or deadlocks in distributed system if the blocking occurs inside event handlers, RPC handlers, or other timing sensitive places in distributed systems, as discussed in Section 2.0.2. A naive solution to this challenge is to move these synchronization primitives around to avoid those regions that are unsuitable for blocking. However, more challenges would come following such movement: the movement may cause unnecessary synchronizations like signal-too-early and lock-too-early, which can also lead to deadlocks; the movement may push lock (signal) and unlock (wait) into different nodes, which can lead to invalid patches.

Strawman-2: failure recovery  Failure recovery rolls back and re-executes a system (component) after failure happens. It may still suffer from the time-out and deadlock problems mentioned above, if repeated re-execution happens within a timing/blocking-sensitive code region. Naively trying to to extend or shorten the re-execution region would encounter many other problems: if the re-execution region is too short, the bug may not be recoverable;
if the re-execution region is too large, it probably will involve rolling back and re-executing multiple nodes’ execution, which is very expensive to guarantee correctness.

3.1.2 Our solutions

To address these challenges, DFix uses a new approach that combines proactive synchronization and reactive failure recovery. DFix comes up with a solution that is a combination of error checking and rollback-retry. The error checking predicts if the bug will happen. If it predicts the bug is going to happen, the rollback-retry will be triggered. The rollback-retry helps to release all the resource that may block the waited operation by restart the program from a reasonable location. And the rollback-retry could bring information for monitor system when rollback happens to avoid mistake decision in monitor.

The above approach works for fixing both atomicity violations and order violations.

**Fixing order violations** Imagine an order violation \((A, B)\), which manifests when \(B\) unexpectedly execute before \(A\).

To fix this bug, DFix would patch the software so that a condition flag \(F\) is set only after \(A\) is executed. This flag can then be checked at a selected location CheckSite before \(B\) is executed. When the software finds that \(F\) is already set at CheckSite, it will proceed to execute \(B\). Otherwise, the software will rollback to an earlier location RestartSite and re-execute, waiting for \(A\) to execute, at which time \(F\) will be set. Figure 3.1 illustrates how this works.

**Fixing atomicity violations** Imagine an atomicity violation \((R, r)\), which manifests when a code region \(R\) executes in parallel with another operation \(r\).

To fix this bug, DFix would patch the software so that a condition flag \(F\) is set when the execution is outside either region and set once the execution enters either region. This flag can then be checked at a selected location CheckSite before the entrance point of \(R_1\). If the checking shows that \(F\) is set, the software will atomically unset \(F\), preventing another
Figure 3.1: This is the overview of different patches for order violation. (a) is the patch based on synchronization primitive. Wait will delay the operation until A is executed. (b) is the failure recovery. It will keep rollback until the failure does not happen. (c) is the patch from DFix. If will keep rollback and predicting if the bug will happen at check site until $F$ is set after A.
Figure 3.2: This is the overview of different patches for atomicity violation. (a) is the patch that protect the atomicity by lock. (b) is the failure recovery. It will keep rollback until the failure does not happen. (c) is the patch from DFix. It protects the atomicity region by a flag $F$. And if CheckSite find $F$ is set, it will rollback.

region from executing in parallel, and continue to $R_1$. If the checking shows that $F$ is unset, indicating the other region is being executed, the software will roll back to RestartSite and re-execute, waiting for the other region’s execution to end, at which time $F$ will be set. Figure 3.2 illustrates how this works.

Road map  The discussion above assumes bug-report inputs that contain xxxxx. We will discuss in more details about how to we obtain such inputs in Section ??.

Given such a bug-report input, to fix it using the above approach, we need to figure out the following components:

- Where is the RestartSite? The suitable location should releases the resource when rollback happens;

- Where is CheckSite? This is the location where predicts if the bug is going to happen.

The code region between restart location and check location is side-effect free.
Figure 3.3: RestartSite search. DFix initializes the RestartSite location at the delayed operation and begins backtrack along the causality chain. First, DFix moves the RestartSite out of the synchronized region. Then, DFix moves the RestartSite to the event enqueue operation because the current RestartSite is inside an event handler. In the end, DFix moves RestartSite the message sending operation, as it finds the RestartSite is in the message handler.

• How to rollback? Unlike using jmp in multi-thread software, rollback in distributed systems may involve multiple threads and even multiple nodes;

• What is the condition flag $F$? The checking condition has to be precise. Imagine we want operation $B$ to wait for $A$. We want $B$ to check for conditions updated by $A$, but not any other operations;

• Where to set and unset the condition flag $F$? Locations right before or after bug instructions may not work.

We discuss these in details in the next section.

3.2 DFix Bug Fixing

3.2.1 RestartSite Selection

As shown in Figure 3.3, given a buggy instruction, like operation $B$ in an order violation or the operation $r$ in an atomicity violation, DFix needs to identify a RestartSite location that serves as the destination of rollback and also the starting point of replay.

Challenges Theoretically, RestartSite can be put at any place that is executed before the buggy instruction. However, many locations are unsuitable for RestartSite due to timeouts
or deadlock concerns. Specifically, we want to (1) put RestartSite outside timing-sensitive regions (e.g., RPC handlers), so that the extra time caused by replays does not cause time-outs; and (2) put RestartSite outside regions that hold deadlock-prone resources such as locks and event-handling threads.

Achieving this goal is challenge. Previous work fixes single-machine concurrency bugs [19, 20] by inserting synchronization operations in the function where the buggy instruction is located or the caller/callee functions. Unfortunately, it is possible that no places inside a call stack are suitable for RestartSite— for example, all the callees of an RPC function are unsuitable to hold an RestartSite.

**Solutions**  DFix searches for the suitable RestartSite location along the causality chain , not the call-stack, where the buggy instruction $I$ is located.

Starting from the location of $I$, denoted as $Loc$, DFix checks the following three conditions to see if $Loc$ is a suitable RestartSite location:

(1) If $Loc$ is inside a lock critical section, DFix considers it as unsuitable for deadlock concerns and moves it to right before the lock acquisition. The lock set is obtained from DCatch which dynamic traces the holding locks for operations.3.3 shows how DFix locates RestartSite in fixing MapReduce-4637[1]. DFix starts with the location $I$. As it is in a synchronized section, DFix will move the candidate RestartSite to before the entrance of this synchronized block.

(2) If $Loc$ is inside an event handler, DFix considers it as unsuitable for deadlock concerns and moves $Loc$ further back along the causality chain to right before the location where the event is put into the event queue. For example, the second step in Figure 3.3 shows that the location before synchronized section is in an event handler. DFix will move it to before the event enqueue operation.

(3) If $Loc$ is inside an RPC handler, DFix considers it as unsuitable for time-out concerns and moves $Loc$ to right before the RPC sending. For example, in Figure 3.3, the location
before event enqueue operation is in a message handler. DFix will move it to before the message sending site.

DFix recursively applies these three rules along the causality chain until a suitable location that passes all the above checkings are found.

Note that, the algorithm above could move the RestartSite location outside the thread where \( I \) is located (i.e., checking-1) or even outside the node where \( I \) is located (i.e., checking-3), which is unique to distributed-system concurrency-bug fixing.

### 3.2.2 CheckSite Selection

CheckSite is the location to check if the bug is going to occur (i.e., whether \( A \) has not executed yet in an order-violation case or whether a critical region is on going now in an atomicity-violation case), and hence whether a rollback needs to be initiated.

**Challenges** Clearly, CheckSite should be put between the RestartSite and the buggy instruction \( I \). However, there is a tradeoff. On one hand, CheckSite cannot be placed too close to the RestartSite and hence too early before the race instruction executes, as it is difficult to predict whether a buggy interleaving or even \( I \) is going to happen too much in advance. On the other hand, CheckSite cannot be placed too close to \( I \) and hence too far away from the RestartSite either, as the difficulty of a correct rollback increases with the growth of the re-execution region.

The concern of re-execution is that the side effect instructions will be executed multiple times which is expected to be once. One solution could be taking a checkpoint at first and restoring the system to the checkpoint the rollback happens. But restoring the whole system to a previous state really takes time. And introducing checkpoint mechanism is not trivial for a software which does not have it before.

**Solutions** DFix decides not to take any checkpoints to keep the patch simple and low-overhead. Consequently, the high level idea of DFix is to search forward from RestartSite
along the causality chain connecting RestartSite and I for the location CheckSite so that (1) the code region between RestartSite and CheckSite can be correctly rolled back and re-executed without any checkpoints (i.e., correctness); and (2) no code region that is a superset of the RestartSite–CheckSite region can satisfy condition-1 (i.e., maximality). Informally speaking, when the forward search hits the first side-effect instruction that is incorrect to roll back and re-execute without checkpointing, CheckSite is decided to be right before that side-effect instruction.

Like that in previous failure-recovery work for multi-threaded software [40], any I/O instructions and heap memory writes are considered as side-effect instructions. Most function-local writes are considered as side-effect free, unless the corresponding function’s callsite is outside the re-execution region. That is, if the whole function stack frame will be rolled back and the function will be re-invoked, updates to the stack variables would not cause any correctness problems. For example in Figure 3.4, the local writes in handler of m1 and e1 is side-effect free. But not for the local writes around rpc call site.

Unlike previous work for multi-threaded software [40], DFix needs to reason about event handling and message sending/receiving. DFix considers an event enqueue operation, an RPC call, and a message sending operation to be side-effect free, if the corresponding event/RPC/message handler is side-effect free. For example, in Figure 3.4, the instruction put(e1) is side-effect free if its handler is side-effect free.

DFix conducts the above analysis considering control flows and calling context to ensure all the possible path reaching the CheckSite is side-effect free.

First of all, DFix clones the causality chain to ensure CheckSite is invoked only in fixing context. The clone copy to causality chain to a new one. DFix puts the CheckSite and RestartSite in new causality, which is only invoked in the patch. The details are explained in Section 3.2.6. Secondly, DFix also guarantees that all the execution paths that follow the new causality chain pass the CheckSite. Otherwise the patch may miss some bug scenarios. DFix builds the control flow graph for each functions in the new function. Then DFix adds
an edge from previous element in causality chain to its function beginning to connect the
control flow graphs. Every path from RestartSite to delayed operation in the connected
control flow graphs is an execution path. Among the instructions shared by all the paths,
DFix locates the furthest one that does not introduce important side effect instruction in all
paths since RestartSite. The CheckSite location is before this shared in instruction.

To filter these important side effect instructions, DFix transfers the binary Java code
to intermediate representation through WALA, a static analysis tool for Java. DFix con-
servatively believes all the field variables are shared. All the field variable writes in the
intermediate representation start with a operand "putfield". DFix filters them based on the
this. For file writes, DFix only consider the log file write as unimportant side effect instruc-
tion. Because more or less entries in log file has no affect on the correctness of the software.
For function or handler analysis, DFix analyze all the execution paths.

3.2.3 Rollback

When the checking fails at CheckSite, DFix-patched software is expected to rollback and
re-execute from RestartSite. How to correctly rollback is an important component of the
patch generation.

Challenges  When RestartSite and CheckSite are inside the same (regular) thread\(^1\), rolling
back a code region that has no side effect is trivial — DFix simple throws an exception from
CheckSite which is caught at RestartSite and put RestartSite into a loop. If the exception
is caught, it will continue the loop.

Unfortunately, RestartSite and CheckSite may be inside different threads, different pro-
cesses, or even different physical nodes, as the RestartSite–CheckSite location design only
make sure that they are along the same causality chain which may span over multiple threads,
processes, and nodes. How to conduct correct rollback goes beyond the scope of traditional

\(^1\). Threads that execute event handlers or RPC handlers are excluded.
single-machine concurrency-bug fixing and failure recovery.

**Solutions** Since rollback within one thread is trivial, our discussion below focuses on rollbacks beyond one thread. We first discuss rollback from CheckSite to RestartSite, when there is only one causality edge between them, and then discuss how to handle more general cases.

There are two types of causality edges, blocking (i.e. RPC calls) and non-blocking (i.e., thread creation, event enqueue-dequeue, and message sending-receiving). When the causality edge between CheckSite and RestartSite is blocking (i.e., RPC call), we can simply discard the work performed before CheckSite in the thread of CheckSite, which is achieved by throwing a special RPC-terminating exception inserted by DFix, and then let the thread of RestartSite to rollback from the RPC callsite to RestartSite through exception handler. For example, In.

When the causality edge is non-blocking, patching faces two tasks. First, DFix needs to turn the causality edge from non-blocking to blocking. That is, originally, the thread of RestartSite would continue its execution right after it initiates the thread of CheckSite through a causality operation \( O_{Cause} \). Now, DFix patch needs to make the RestartSite thread waits right after \( O_{Cause} \) until the checking succeeds at CheckSite. Otherwise, correct rollbacks cannot be guaranteed. Second, DFix needs to allow CheckSite to unblock the thread of RestartSite, either allowing it to proceed when the checking passes or to rollback when the checking fails.

Both tasks are accomplished by introducing a communication flag variable \( V \). DFix inserts a \( V \)-checking function right after \( O_{Cause} \) in the thread of RestartSite, and a \( V \)-setting operation at CheckSite, as shown in Figure 3.4 (T2). After the event enqueue operation, it checks the shared variable \( V \). The check passes only when the event handler pass the CheckSite and set \( V \). When CheckSite and RestartSite belong to the same process, \( V \) is simply a new heap object. When they belong to different processes, DFix creates a heap object.
that belongs to the process of RestartSite and then make CheckSite updates through
RPCs. Figure 3.4 show an example that first rollback from event handler to RPC hander
and then from RPC handler to the RPC sending. In the first rollback, DFix introduces a
new synchronization: f.wait(). In the second rollback, DFix utilizes the existing mechanism.

Now the only problem is that the new synchronization for rollback may be inside the
timing-sensitive region in Section 3.2.3. To avoid long time wait, DFix uses a timed wait.
If the wait is timeout, the rollback is triggered automatically. In Figure 3.4, it throws an
exception for rollback if the f.wait() is timeout. If the first e1 waits in the process queue
too long, e1 will be put in the queue multiple times because the f.wait() keeps timeout. To
guarantee the correctness, if one e1 pass the CheckSite which means this is a correcting, all
the following e1 instances will be dropped. DFix implements a causality clone to ensure the
dropped event are all duplicate which will be explained in section 3.7.1.

Every time, DFix consider the bug under a specific causality chain. If one bug involves
multiple causality chain, DFix fixes them independently. And the causality clone ensures no
interference between the patches of different causality chain patches.

3.2.4 Creating condition flags

As discussed in Section 3.1, a condition flag $F$ is needed to fix every bug. This flag is checked
at CheckSite. When it is set, the software can proceed to execute operation $B$, in case of an
order-violation patch, or enter a mutual-exclusive region $R$, in case of an atomicity-violation
path. When it is unset, the software will rollback to RestartSite and re-execute. In this
sub-section, we discuss how DFix creates such a condition flag $F$ in its patch.

Challenges There are two challenges in this task. First, when DFix needs to fix multiple
dCbugs in a software project, it needs to create different flag variables for different bugs.
Second, there might be multiple dynamic instances of a static instruction, and we may
need to enforce ordering relationship for specific dynamic instance(s) of a static instruction.
Figure 3.4: Example about rollback from inside an event handler to rpc callsite int thread T1. After event enqueue operation put(e1) in thread T2, there is a timed wait to make it synchronized. f.wait() wait f is set somewhere. f.set(v) sets a specific value v to f. If timeouts, it will throw an exception to rpc callsite. And the wait is signaled after check passes in CheckSite in thread T3. To avoid processing multiple same e1, DFix adds a check-drop at the beginning of its handler.
Consequently, the condition flag $F$ has to be able to differentiate not only different bugs but also different dynamic instances of the same static instruction.

Note that the second challenge above has not been addressed by previous single-machine concurrency-bug fixing tools [19, 20], where all dynamic instances of a static instruction are treated the same. However, we have found it to be crucial in fixing DCbugs. Ignoring it can easily lead to either incomplete bug fixing or deadlocks. In Figure 2.1, all the container in MapReduce will send the read message to read map entry based on its own task ID in AM. Enforcing order between the Cancel Task and an arbitrary read cannot guarantee fix the bug.

**Our Solutions** At high level, to fix each bug, DFix creates a map of flags, with different flags updated and checked for different dynamic instances of $A$, $B$, CheckSite, etc.

For an order violation $A$–$B$, the DFix patch conducts flag updates right after the execution of $A$. The memory address touched by $A$, represented by heap object hash-code together with field offset, will be used as the key to look up the flag map, and be used to differentiate different dynamic-instances/flags. Similarly, the corresponding $B$ instance’s CheckSite will use the memory address touched by $B$ to look up corresponding flag in the flag map.

For an atomicity violation $R_1$–$R_2$, the DFix patch conducts flag updates right after the program enters region $R_1$ or $R_2$. DFix uses the memory address of the beginning of region to look up the flag map. DFix patch also reset the flag after the exit of atomicity regions.

---

Figure 3.5: the rectangles are instance of $A$. Circles are instances of $B$. White operations access the same memory location. Three grays access another one. Bring order across white and gray is helpless for fixing. Associating the ordered operations with address provides better order control.
memory address of the end of region is used as key to look up the flag map. Actually the interesting conclusion is that all the atomicity regions can be implemented with a beginning and an end which access the same memory location. So the update and reset can be matched. Before the program enters the region, the CheckSite related to this region will check if there are an active competing region.

Of course, sometimes, buggy instructions touch shared heap object through API calls. For example, A might be `map.set(key)` and B might be `map.get(key)`. For API related invocation, it is not easy to know the explicit address it In these cases, instead of directly using address, DFix concatenates the hashcode of invoked object and the parameters as the map key.

The next question is how DFix patch can compute the map key (i.e., object touched by buggy instructions) at the update location and at the CheckSite. It is trivial to compute the map key after the update operation as it is already executed.

DFix conducts this through causality slicing. Program slicing[38] is selecting a subset statements, the slice, that may affect the value at point of interest from provided program statements. Because we want to compute the identity in advance in the causality chain. We only need to know which instructions in causality chain may affect the value of racing operation. Based on different provided program statements, existing program slicing are classified into full program slicing and callstack slicing. But none of them can be applied to causality slicing. Callstack slicing calculates the affect instructions along the callstack. But callstack misses the causality information. Even full program slicing involves the causality information, the overhead is too huge. We test that full program slicing of one interested point in MapReduce takes over 60G memory and cannot finish over 2 days.

Unlike the existing slicing technologies, DFix collects the instructions along the causality chain as the provided set. To implement slicing, DFix utilizes the slice computation API from WALA, a Java static analysis tool. Slicer API in WALA can return the slice of the pointed object from its callstack.
To compute the causality slice, DFix starts with callstack slice of racing operation. If the racing operation depends on the event or message object which cause this operation, the callstack slice of this event object at enqueue location or message object in sending location will be added to causality slice. DFix keeps adding the parent event or message callstack slice until no more instructions added. Then DFix removes the instructions that are not in the causality chain to reach the final causality slice.

DFix computes the identity of racing operation at the CheckSite. To construct the computation, DFix collects the slice of each component of the identity. At the CheckSite, DFix executes these slices to calculate the hashcodes and concatenate them together for identity. To invoke the instructions in CheckSite, DFix adds a prefix to each variable that is accessed outside its original scope. For example, a instruction in slice is temp=f(a) in class A. To execute it at CheckSite, DFix changes it to temp=A.read(A.a). Based on the computed identity, the patch access the correct flag to decide if rollback.

Correctness

The condition to trigger rollback is a buggy timing check based on slice. The slice which computes the identity of data race may read some shared variables which could be modified by other threads. That means the the precomputed identity may be different with the exact wait operation as the environment changes in between.

In general, it is difficult to prove that the computed identity never change between CheckSite and delayed operation. But in some cases, DFix can prove every time the precomputed identity is the same as exact identity. For example, the wait operation accesses a global variable. The global variable is located when the process starts. Its address never changes during runtime. DFix looks through the read operation in the slice. And if all the read variables are unchanged, the computed identity will be unchanged. DFix consider the following variables as unchanged:(1) static variables. (2) the local variables in RestartSite. (3)The class instances that contains the synchronization for rollback from RestartSite to CheckSite.
along the causality chain. RestartSite is waiting the checking result from CheckSite. No one can change its local variables. In Java, the hashcode of one class never change if its method is active. Along the causality chain, DFix ensure some methods keep active for rollback. The super classes of these active methods never changes.

If the precomputed identity is wrong, it leads unnecessary rollback or mistakenly continue execution. For unnecessary rollback, it will finial compute the correct identity after the heap environment is set. This brings extra overhead without hurting the correctness of patch. For mistakenly release the execution, it is relative rare. Usually the identity is a hascode or combination of hashcode and value. It is rare to mistakenly compute a identity that just pass the check.

As distributed system involves so many nondeterminism, it is challenge to predict the buggy timing 100 percent correct in patch. But incorrect prediction in DFix does not bring new logic or crash to the software. It only causes more overhead to fix the bug or hardly cannot fix the bug.

In addition to correctness, the identity computation code should also be side effect free, as it will be executed multiple times. If neither correctness or side effect checking is not passed, DFix would assign the ID with a constant value to provide coarse granularity control.

3.2.5 Deadlock Analysis

As we mentioned in section 3.5.1, the identity addition for order violation happens after the racing operation that should occur first. The removal for atomicity violation occurs after the end of atomicity region. But there is no guarantee that these operations always happen, which could cause deadlock. DFix tries prove the occurrence of these operation. If fail, DFix would limit rollback times to avoid deadlock.

In distributed system, the disappearance of operation has two reasons: related to failure or not. The failure related reason could be the message carrying signal operation for order enforcement id dropped because of the net work issue or the node sending this message
crashes. As the failure could happen at any time, the only solution to avoid deadlock is timeout. The timeout is implemented as recording how many time the rollback happens in patch. If it reaches the limit time, patch will assume the waited operation will never happen. In the timeout handler, it will pass the CheckSite.

The failure unrelated reason is that the change of input case or different schedule with the same input case. Even already using timeout, DFix still explores if the timeout is necessary for no failure situations. DFix analyzes the source code to try to provide some proof about when the signal operation always happens. These proofs are all build under the assumption that the distributed system does not meet failures in this execution.

If no failure, DFix analyzes if A always happens under the occurrence of B for order violation (B,A). It also check if c always happens under the occurrence of p for atomicity violation (p,c,r). p is the beginning of atomicity region, where c is the end. To do that, DFix searches an instruction set whose element dominates op1. If a dominates b, all the paths to b must pass a. In other words, a’s occurrence indicates b’s occurrence. So all the instruction from this domination set must happen since knowing the op1 happens. If op2 is in this set, the proof is done. Otherwise, DFix collects another set whose instruction is post-dominated by op1. If a post-dominates b, all the exit paths from b pass a. a’s occurrence can be referred from b’s occurrence. If any instruction in the post-domination set happens, op2’s occurrence can be reffed. Sometime the post-domination set contains the enter of main function. That means the op2 always happens even without knowing the information about op1. If the domination set and post-domination set overlap, op2’s will always happen since op1’s occurrence. To collect the domination and post-domination set through inter-procedure analysis, DFix first inherits the rules from single machine software[19, 20]. DFix second apply the rules related to causality path:(1) event enqueue operation dominates the event handler. The event handler post-dominates the enqueue operation; (2) message sending site dominates the message handler. message handler post-dominates the message sending site; (3) third party mechanism. Like updating ZooKeeper dominates the auto-forwarding
Figure 3.6: Domination and post-domination analysis. For operation F, A,B,C dominate it. For operation C, it is post-dominated by F. A is an event enqueue operation. B is the beginning of event handler. So A always dominates B. And B post-dominates A.

message handler relation to this update. Figure 3.6 shows an example about the domination and post-domination analysis.

For the situations that cannot prove op2 always happen, DFix uses timeout rollback to avoid the deadlock. Some other tools [19, 20] would statically analyze some locations they are pretty sure that the signal operation never show up. And they place extra lock release operations on these locations to avoid triggering timeout naively, which is call safety-net design. But in distributed system, it is challenging to know when one operation will not show up anymore. And sometime, like heartbeat and gossip, some operations keep happening. It is difficult to figure out if one operation in heartbeat or gossip won’t be executed anymore. DFix looks through causality of op2. If op2 is inside an event or message handler, timeout is the better solution than safety-net design. The original safety-net is implemented through single node static analysis. But this kind of analysis usually make mistakes in distributed system because of missing information from other nodes.
3.2.6 Others

Causality Clone

DFix clones functions to ensure two things: (1) All delayed operation that may cause the bug follow the cloned causality chain (2) each patch from DFix only takes effect under certain causality context. All the patches occur in the cloned function. For order violation \((P1,P2)\), DFix clones the causality for both \(P1\) and \(P2\). For atomicity violation \((p,c,r)\), DFix only clones the causality chain for all the operations.

Consider a causality path \((c_0,f_0,l_0) \rightarrow (c_1,f_1,l_1) \rightarrow ... \rightarrow (c_n,f_n,l_n)\). DFix clones every function \(f_k\) in the causality path to \(f'_k\) in the class \(c_k\). If the \(\rightarrow\) before \((c_k,f_k,l_k)\) is a function call, DFix will find the same call in \(f'_{k-1}\) and change the target function to \(f'_k\). If the \(\rightarrow\) is a thread creation, DFix will clone a new class \(c'_k\) and set it to the parameter in corresponding thread creation in \(f'_{k-1}\). If the \(\rightarrow\) is an event enqueue, DFix will assign the event in \(f'_{k-1}\) a new type and bind \(f'_k\) to new event type handler. In other words, causality create a totally new path which is only invoked in target context. All the patches happen in this new path. The reason for choosing causality path clone instead of normal function callstack clone is that causality path is more effective to differentiate the operation.

As one event could be put into processing queue multiple time if there is an asynchronized instruction, DFix needs to differentiate the event created because of rollback and the original enqueue operation call. Or a new enqueued event may be dropped if the patch treats it as the duplicate events generated by rollback. To keep the difference, DFix clones the the causality path twice. DFix not only clones a \(f'_k\) from \(f_k\), but also clones a \(f''_k\). The connection for \(f''_k\) is still same with what is mentioned in last paragraph. The only extra thing is call \(f''_k\) if program rolls back to \(f'_k\). In this way, DFix separates a new invocation and the repeat. DFix is able to put different action for these different action. Figure 3.7 shows an example about double clone. DFix only put the already processed check drop for duplicate events in the second cloned causality path.
Figure 3.7: If only clone once, it is hard to differentiate e1 is a repeat event or a new event from another invocation. DFix clones it twice with another path that contains '1'. All the repeat events happen in the second path. In this way the new event will not be dropped mistakenly.
CHAPTER 4
EVALUATION

DFix is implemented using WALA v1.3.5 and JavaParser v3.2.4 for a total KLOC. Below are more details.

4.0.1 Methodology

4.1 representative of these benchmarks. We evaluated DFix on seven real-world distributed concurrency bugs from four popular open-source cloud systems: Hadoop Mapreduce huge data processing platform (MR); ZooKeeper centralized metadata management service (ZK); Cassandra scalable key-value storage system (CA); HBase distributed key-value stores (HB). The developer fixing times range from several days to 4 month.

We obtain these benchmarks from DCatch’s detection results[29]. They are all triggered by unexpected timing of inter-node communication. They also cover the different root cause: atomicity violation (AV) and order violation (OV). The number of corresponding DCatch reports varies from 1 to 7. Each reports means a data race. Only order violation could have one corresponding data race which is exactly the root cause. For ZK-1270, it has 4 data races that all cause the same problem. For atomicity violation, it really involves multiple data races. The reason is that the boundary of atomicity region corresponds to two data races. The other data races includes the operations inside the atomicity region.

We setup the testing environment on two machines cluster. Each computer has Intel RCoreTM i7-3770 and 8GB of RAM. Both of them use Ubuntu 14.04 and JVM v1.7. And

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>LoC</th>
<th>Workload</th>
<th>Manual Fixing</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA-1011</td>
<td>61K</td>
<td>Startup</td>
<td>114 days</td>
<td>AV</td>
</tr>
<tr>
<td>HB-4539</td>
<td>188K</td>
<td>Split &amp; Alter table</td>
<td>5 days</td>
<td>AV</td>
</tr>
<tr>
<td>HB-4729</td>
<td>213K</td>
<td>Enable table &amp; expire server</td>
<td>27 days</td>
<td>AV</td>
</tr>
<tr>
<td>MR-3274</td>
<td>1.2M</td>
<td>Startup + WordCount(WC)</td>
<td>4 days</td>
<td>OV</td>
</tr>
<tr>
<td>MR-4637</td>
<td>1.3M</td>
<td>Startup + WordCount(WC)</td>
<td>48 days</td>
<td>OV</td>
</tr>
<tr>
<td>ZK-1144</td>
<td>102K</td>
<td>Startup</td>
<td>8 days</td>
<td>OV</td>
</tr>
<tr>
<td>ZK-1270</td>
<td>110K</td>
<td>Startup</td>
<td>7 days</td>
<td>OV</td>
</tr>
</tbody>
</table>

Table 4.1: DFix Benchmarks.
Table 4.2: DFix Overall Result. M-DL means the deadlock caused by message block. E-DL means the deadlock caused by event block. L-DL is the deadlock caused by lock. X means no naive solution.

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>NoFix</th>
<th>DFix</th>
<th>#Roll</th>
<th>NaiveFix</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA-1011</td>
<td>0.6</td>
<td>0</td>
<td>0.7</td>
<td>M-DL</td>
</tr>
<tr>
<td>HB-4539</td>
<td>1</td>
<td>0</td>
<td>2.2</td>
<td>X</td>
</tr>
<tr>
<td>HB-4729</td>
<td>0.6</td>
<td>0</td>
<td>5.7</td>
<td>X</td>
</tr>
<tr>
<td>MR-3274</td>
<td>0.8</td>
<td>0</td>
<td>0.7</td>
<td>E-DL</td>
</tr>
<tr>
<td>MR-4637</td>
<td>0.5</td>
<td>0</td>
<td>0.5</td>
<td>E-DL</td>
</tr>
<tr>
<td>ZK-1144</td>
<td>0.8</td>
<td>0</td>
<td>0.8</td>
<td>L-DL</td>
</tr>
<tr>
<td>ZK-1270</td>
<td>1</td>
<td>0</td>
<td>0.8</td>
<td>L-DL</td>
</tr>
</tbody>
</table>

The machines are connected through Ethernet cable. To show the fixing ability of DFix, we prepare the bug input for each benchmark, shown in the 4.1. Beside the workload for triggering the bugs, we also modify the source code by inserting some sleep to increase the bug exposing probability.

4.0.2 Overall Result

Table 4.2 shows the overall results of DFix. Before testing, DFix inserts a random sleep to increase the bug exposed rate. DFix tests the software under three situations: (1) Original software. (2) Patch the software by DFix. (3) Patch the software naively with synchronization primitive. DFix runs each benchmark ten times and calculates the probability of bug occurrence. None of bugs in benchmarks occurs under the patch of DFix. And naive patch leads deadlock in 5 benchmarks. In CA-1011, the deadlock is caused by the delay in message handler blocking other message. In two ZK benchmarks, the deadlock is caused by keeping holding the lock. In two MR benchmarks, naive solution keeps holding the event process handler which cause the deadlock. The remain two do not have naive fix solution because the racing operations are in different nodes. The average rollback time varies from 0.5 to 6.
### Table 4.3: DFix Bug Understanding.

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>Output</th>
<th># Input Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA-1011</td>
<td>2 Atomicity regions</td>
<td>7</td>
</tr>
<tr>
<td>HB-4539</td>
<td>1 Atomicity region vs 1 op</td>
<td>3</td>
</tr>
<tr>
<td>HB-4729</td>
<td>2 Atomicity regions</td>
<td>6</td>
</tr>
<tr>
<td>MR-3274</td>
<td>1 root cause data race</td>
<td>1</td>
</tr>
<tr>
<td>MR-4637</td>
<td>1 root cause data race</td>
<td>1</td>
</tr>
<tr>
<td>ZK-1144</td>
<td>1 root cause data race</td>
<td>1</td>
</tr>
<tr>
<td>ZK-1270</td>
<td>1 root cause data race</td>
<td>4</td>
</tr>
</tbody>
</table>

#### 4.0.3 Understanding Result

Table 4.3 shows the understanding results of DFix. First, all the bug type are detected correctly. For order violation, ZK-1270 involves 4 bug reports. And DFix could find the true root cause among these 4 bug reports. For atomicity violation, DFix figures out two atomicity regions competing with each other for CA-1011 and HB-4729. For HB-4539, DFix only finds one operation competing with atomicity region. All these advanced information related to this bug is same as the bug report in website except HB-4539. In reality, HB-4539 is two atomicity regions competing with each other. The reason is that DCatch only provides the testing result of data races that do not share the same callstack. For the data races that own the same callstack, DCatch random pick one to test. This information loss may miss useful data race that share callstack but not the causality path. The overall understanding result is quite close the human and the time cost is all less than 12 seconds.

#### 4.0.4 Patch details

Table 4.4 shows the the details of every patch. Each patch has different rollback region. 3(CA,HB-3539,HB-4729) of them need to rollback from a blocking message handler to its sender. 2(MR-4637,MR-3274) of them rollbacks from event handler to the message sender whose handler enqueues the event. The remain 2(ZK-1144,ZK-1270) rollback out of the synchronized block.
<table>
<thead>
<tr>
<th>Bug ID</th>
<th>Rollback</th>
<th>ID SE Free</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA-1011</td>
<td>BlockingM</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HB-4539</td>
<td>BlockingM</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HB-4729</td>
<td>BlockingM</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MR-3274</td>
<td>BlockingM,E</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MR-4637</td>
<td>BlockingM,E</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ZK-1144</td>
<td>SyncBlock</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ZK-1270</td>
<td>SyncBlock</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.4: DFix Patch Details.

The IDs computed for deciding rollback are all proved unchanged during patching. 2(HB-4539, HB-4729) of them access the static variable whose address never change during process runtime. The remaining five are proved by the address of field variable does not change when one of the function of it container class is running.

All the rollback regions are side effect free. For 5 benchmarks where rollback region contains event enqueue operation or message sending, it is side effect free because the handler of the event or message is side effect free.

In 5 of benchmark, the signal operation or end of the atomicity region is guaranteed to show up. In MR-3274, the signal operation post-dominates the begin of main function. As long as the process enters the main function, the signal operation in MR-3274 occurs. In HB-4539 and HB-4729, the atomicity region is from the update to ZooKeeper to the forwarded event after it finishes the update. ZooKeeper guarantees that it always forward the event to notify the update to other processes. In ZK-1144 and ZK-1270, its signal operation post-dominates an instruction that dominates the wait operation. If the wait operation occurs, the signal operation mush happen.

4.0.5 Performance.

Table 4.5 shows the general overhead in correct execution caused by DFix ’s patches. Comparing with Table 4.2, the overhead is tested without inserting any sleep. DFix runs the tests 5 times and calculates the average finish time of workload. All the overheads are below
<table>
<thead>
<tr>
<th>Bug ID</th>
<th>Baseline</th>
<th>Fixed</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA-1011</td>
<td>16.66s</td>
<td>16.74s</td>
<td>0.5%</td>
</tr>
<tr>
<td>HB-4539</td>
<td>9.58s</td>
<td>9.60s</td>
<td>0.2%</td>
</tr>
<tr>
<td>HB-4729</td>
<td>36.69s</td>
<td>36.72s</td>
<td>0.1%</td>
</tr>
<tr>
<td>MR-3274</td>
<td>25.20s</td>
<td>25.31s</td>
<td>0.4%</td>
</tr>
<tr>
<td>MR-4637</td>
<td>11.02s</td>
<td>11.1s</td>
<td>0.7%</td>
</tr>
<tr>
<td>ZK-1144</td>
<td>0.67s</td>
<td>0.67s</td>
<td>0%</td>
</tr>
<tr>
<td>ZK-1270</td>
<td>0.72s</td>
<td>0.72s</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 4.5: DFix Patch Performance.

1 percent. When the timing is correct, the patch only introduces several extra memory access, which does not lead too much overhead. In Mapreduce, the patch will make the bug related event wait for the result from CheckSite. It leads a little more overhead than other benchmarks.
CHAPTER 5
RELATED WORK

Bug Detection in Distributed Systems  Recently DCBugs draws increasingly more attention in research. Distributed system model checkers(or “dmck” in short) have many inspiring tools[15, 22, 25, 36, 27] to expose the DCBugs. Dmck collects all the nondeterministic events and permuting their ordering to test the software. The nondeterministic events in distributed system includes failures, message arrivals, and the local computation. Beside dmck, data race detection tool[29] also helps to find the DCBugs. It collects the memory access during runtime and build a happen before graph. Based on the graph, it finds all the data races and filter the harmful one through static analysis.

Improving the Reliability of Distributed Systems  Program verification[16, 39] is a strong method to provides reliability of software. It does not have false negatives or positives. The problem of verification is it needs a long proof. Even for basic protocol implementation, it need thousands of line code for implementation.

For reliability, there are built-in fault tolerance exists in all distributed systems. These tolerance mechanisms most handle the failure situations like node crash, and network traffic. But it cannot handle scenarios when all the components are alive but a DCbug occurs. And a bug-free tolerance mechanisms are not trivial to implement. The failure recovery leads bugs because of bad implementation. We still need to bug fix tool to actually achieve reliability of software.

Automated Bug Fixing  For concurrency bug fixing, there are multiple interesting fixing strategies. [19, 20] fix concurrency bugs by introducing synchronization primitive for better timing control. [28, 5, 6] changes the timing of software through swapping order of program statements. But all of them only work for single machine software. The various synchronization in distributed system cause deadlock under their patch.
Many techniques have been proposed to automatically fix general bugs\cite{7, 12, 21, 23, 30, 34}. They neither assume the unique feature nor knowledge of the bug, which cannot be satisfied by distributed systems.
CHAPTER 6

CONCLUSION

We have described DFix, a framework for automatically fixing concurrency bugs in distributed system. We implemented and evaluated DFix in seven real world bugs. And the results show that DFix fix all the bugs without introduces new functionality and huge performance loss. The fixing strategy of DFix avoid the deadlock and timeout under the various synchronization in distributed system. The static analysis conducted by DFix provides finer granularity timing control which is critical for DCBugs fixing.
REFERENCES


