DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems

Haopeng Liu Guangpu Li Jeffrey F. Lukman Jiaxin Li
Shan Lu Haryadi S. Gunawi Chen Tian*

University of Chicago *Huawei US R&D Center

{haopliu, cstjygpl, lukman, jiaxinli, shanlu, haryadi}@cs.uchicago.edu chen.tian@huawei.com

Abstract

In big data and cloud computing era, reliability of distributed systems is extremely important. Unfortunately, distributed concurrency bugs, referred to as DCbugs, widely exist. They hide in the large state space of distributed cloud systems and manifest non-deterministically depending on the timing of distributed computation and communication. Effective techniques to detect DCbugs are desired.

This paper presents a pilot solution, DCatch, in the world of DCbug detection. DCatch predicts DCbugs by analyzing correct execution of distributed systems. To build DCatch, we design a set of happens-before rules that model a wide variety of communication and concurrency mechanisms in real-world distributed cloud systems. We then build runtime tracing and trace analysis tools to effectively identify concurrent conflicting memory accesses in these systems. Finally, we design tools to help prune false positives and trigger DCbugs.

We have evaluated DCatch on four representative open-source distributed cloud systems, Cassandra, Hadoop MapReduce, HBase, and ZooKeeper. By monitoring correct execution of seven workloads on these systems, DCatch reports 32 DCbugs, with 20 of them being truly harmful.

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

Keywords Concurrency Bugs, Distributed Systems, Bug Detection, Cloud Computing

ASPLOS '17, April 08-12, 2017, Xi'an, China
© 2017 ACM. ISBN 978-1-4503-4465-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037735

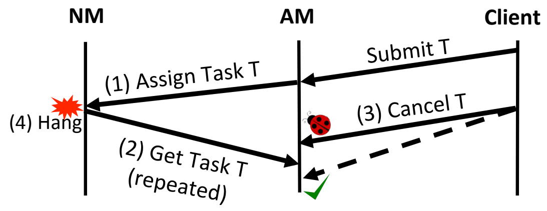


Figure 1. A Hadoop DCbug: Hang (buggy) if #3 happens before #2, or no failure (√) if the otherwise.

1. Introduction

1.1 Motivation

In big data and cloud computing era, distributed cloud software infrastructures such as scale-out storage systems [4, 6, 11, 33], computing frameworks [5, 30], synchronization services [3, 18], and cluster management services [16, 43], have emerged as a dominant backbone for modern applications. Users expect high reliability from them, which unfortunately is challenging to guarantee due to wide-spreading software bugs [12–14, 47].

Among all types of bugs in distributed systems, distributed concurrency bugs, referred to as DCbugs, are among the most troublesome [13, 24]. These bugs are triggered by untimely interaction among nodes and could propagate the resulting errors beyond one node. Previous studies have shown that DCbugs widely exist in real-world distributed systems, causing a wide variety of failure symptoms like data corruptions, system crashes, job hangs, etc [13, 14, 23, 24, 47].

Figure 1 illustrates a real-world DCbug from Hadoop MapReduce. It is triggered by unexpected timing among Node-Manager (NM), Application-Manager (AM), and the client nodes. Specifically, after the AM assigns a task T to a container in NM (#1), this NM container tries to retrieve the content of task T from AM (#2). However, when this retrieval request is delivered to AM, task T has already been canceled upon the client's request (#3). Not anticipating

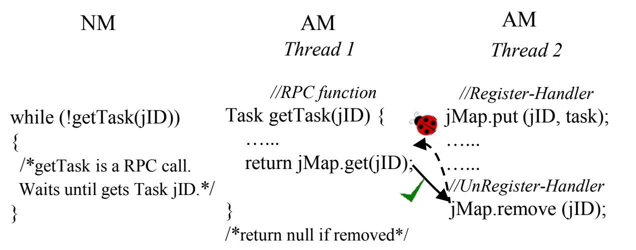


Figure 2. Root cause of the DCbug shown in Fig. 1

this timing scenario, the NM container hangs (#4), waiting forever for AM to return task T to it.

As we can see, DCbugs are non-deterministic and hide in the huge state space of a distributed system spreading across multiple nodes. They are difficult to avoid, detect, and debug.

There are only a few sets of approaches that tackle DCbugs, to the best of our knowledge: software model checking, verification, verifiable language, record and replay debugging. Although this set of techniques are powerful, they suffer from inherent limitations. Distributed system model checkers [14, 20, 23, 41, 47] face state-space explosion problems, making them difficult to scale for many large real-world systems. Verification approaches [15, 45] require thousands of lines of proof to be written for every protocol. Verifiable language [7] is not deployed, as low-level imperative languages are still popular for performance reasons. Record and replay techniques [25] cannot help discover bugs until software has failed and are not yet effective for debugging DCbugs due to the huge number of timing-related events in distributed systems.

In comparison, there is one approach that has been widely studied for combating local concurrency (*LC*) bugs in single-machine software but has yet been explored for DCbugs — *dynamic bug detection* [9, 17, 19, 26, 27, 37]. In a nutshell, dynamic bug-detection techniques monitor and analyze memory accesses and synchronization operations, and identify conflicting and concurrent memory accesses as LCbug suspects. *Conflicting* means that multiple accesses are touching the same memory location with at least one write access. *Concurrent* means that there is no *happens-before* causality relationship between accesses, and hence accesses can happen one right after the other in any order [21]. These techniques do not guarantee finding all bugs and often report many false positives. However, they can usually work directly on large existing real-world systems implemented in popular languages, without much annotation or code changes from developers.

Despite its benefits, bug-detection approach has not permeated the literature of combating DCbugs. Thus, in this paper, we present one the first attempts in building DCbug-detection tool for distributed systems.

1.2 Opportunities and Challenges

Our attempt of building a DCbug detection tool is guided by our following understanding of DCbugs.

Opportunities DCbugs have fundamentally similar root causes as LCbugs: unexpected timing among concurrent conflicting accesses to the *same* memory location inside *one* machine. Take the DCbug in Figure 1 as an example. Although its triggering and error propagation involve communication among multiple nodes, its root cause is that event handler UnRegister could delete the jID-entry of jMap concurrently with a Remote Procedure Call (RPC) getTask reading the same entry, which is unexpected by developers (Figure 2).

This similarity provides opportunities for DCbug detection to re-use the theoretical foundation (i.e., happens-before ordering) and work flow of LCbug detection. That is, we can abstract the causality relationship in distributed systems into a few happens-before (HB) rules; we can then follow these rules to build an HB graph representing the timing relationship among all memory accesses; finally, we can identify all pairs of concurrent conflicting memory accesses based on this HB graph and treat them as DCbug candidates.

Challenges DCbugs and distributed systems also differ from LCbugs and single-machine systems in several aspects, which raise several challenges to DCbug detection.

1. More complicated timing relationship: Although root-cause memory accesses of DCbugs are inside one machine, reasoning about their timing relationship is complicated. Within each distributed system, concurrent accesses are conducted not only at thread level but also node level and event level, using a diverse set of communication and synchronization mechanisms like RPCs, queues, and many more (exemplified by Figure 2). Across different systems, there are different choices of communication and synchronization mechanisms, which are not always standardized. *Thus, designing HB rules for real-world distributed systems is not trivial. Wrong or incomplete HB modeling would significantly reduce the accuracy and the coverage of DCbug detection.*

2. Larger scales of systems and bugs: Distributed systems naturally run at a larger scale than single-machine systems, containing more nodes and collectively more dynamic memory accesses. DCbugs also operate at a larger scale than LCbugs. For example, the DCbug shown in Figure 1 involves three nodes (client, AM, and NM) in its triggering and error propagation. *The larger system scale poses scalability challenges to identify DCbugs among huge numbers of memory accesses; the larger bug scale also demands new techniques in bug impact analysis and bug exposing.*

3. More subtle fault tolerance: Distributed systems contain inherent redundancy and aim to tolerate component failures. Their fault-tolerance design sometimes cures intermediate errors and sometimes amplifies errors, making it difficult to judge what are truly harmful bugs. For example, in Figure 2, the jMap.get(jID) in Thread-1 actually executes concurrently with two conflicting accesses in Thread-2: jMap.put(jID,task) from the Register handler, and jMap.remove(jID) from the UnRegister handler. The former is *not* a bug, due to the re-try while loop in NM; the latter is indeed a bug, as it causes the re-try while loop in NM to hang. *Thus, the subtle fault tolerance features pose challenges in maintaining accuracy of DCbug detection.*

1.3 DCatch

Guided by the above opportunities and challenges, we built DCatch, to the best of our knowledge, a pilot solution in the world of DCbug detection. The design of DCatch contains two important stages: (a) design the HB model for distributed systems and (b) design DCatch tool components.

HB Model: First, we build an HB model on which DCatch will operate, based on our study of representative open-source distributed cloud systems. This HB model is composed of a set of HB rules that cover inter-node communication, intra-node asynchronous event processing, and intra-node multi-threaded computation and synchronization. The details will be discussed in Section 2.

DCatch tool components: Next we build DCatch, our DCbug-detection tool. Although it follows the standard work flow of many LCbug detectors, our contribution includes customizing each step to address unique challenges for DCbugs.

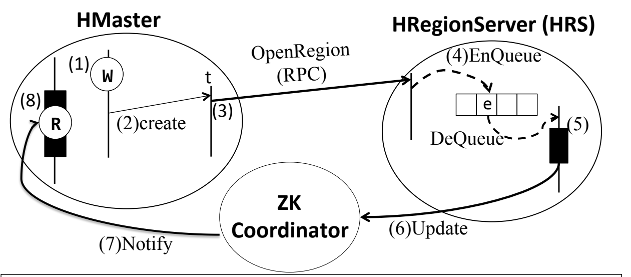
1. Run-time tracer traces memory accesses, event operations, inter-node RPCs, socket communication, and others as the system runs. The scope and granularity of this component is carefully designed to focus on inter-node communication and computation, which helps us to address the large-scale challenge in DCbug detection and make DCatch scale to large real-world distributed cloud systems (Section 3.1).

2. Offline trace analysis processes run-time traces to construct an HB graph for all recorded memory accesses and reports all pairs of concurrent conflicting accesses as DCbug candidates. Our contribution is the implementation of DCatch HB model for real-world distributed systems (Section 3.2).

3. Static pruning analyzes the program to figure out what might be the local and distributed impact of a DCbug candidate. It estimates which DCbug candidates are unlikely to cause failures, avoiding excessive false positives (Section 4).

4. DCbug triggering re-runs the system and manipulates the timing of distributed execution according to the bug report, while considering the diverse concurrency and communication mechanisms in distributed systems. It helps trigger true bugs and further prunes false positives (Section 5).

We evaluated DCatch on 4 varying real-world distributed systems, Cassandra, HBase, Hadoop MapReduce, and ZooKeeper. We tested 7 different workloads in total on these systems. Users have reported timing-related failures under these workloads. DCatch reports 32 DCbugs. With the help of



W: `regionsToOpen.add(region)`
R: `if(regionsToOpen.isEmpty())`
(After this **R**, it has another **W** (`regionsToOpen.remove(region)`))

Figure 3. HBase guarantees W to execute before R through a wide variety of causality relationships. (1) HMaster adds a region to regionsToOpen list (i.e., the W); (2) a thread t is created to open the region; (3) t invokes an RPC call OpenRegion; (4) The RPC implementation in HRS puts a region-open event e into a queue; (5) e is handled, at the end of which (6) a request is sent to ZooKeeper to update the status of the corresponding region to be RS_ZK_REGION_OPENED; (7) ZooKeeper sends a notification to HMaster about this state change; (8) while handling this notification, the event-handler in HMaster reads regionsToOpen (i.e., the R).

DCatch triggering component, we confirmed that 20 out of these 32 DCbugs are indeed harmful: 12 of them explain the 7 failures we were aware of and the remaining 8 could lead to other failures we were unaware of. The detailed experimental results are presented in Section 7.

2. DCatch Happens-Before (HB) Model

Goals & Challenges Timing relationship is complicated in distributed systems. For example, to understand the timing between R and W in Figure 3, we need to consider thread (step 2 in Figure 3), RPC (step 3), event handling (step 4 & 5), ZooKeeper synchronization service (step 6 & 7), etc. Missing any of these steps will cause R and W to be incorrectly identified as concurrent with each other.

Our goal here is to abstract a set of HB rules by studying representative distributed cloud systems. Every rule R represents one type of causality relationship between a pair of operations[1], denoted as $o_1 \overset{R}{\Rightarrow} o_2$. These rules altogether allow reasoning about the timing between any two operations: if a set of HB rules chain o_1 and o_2 together $o_1 \overset{R^1}{\Longrightarrow} oo_1 \overset{R^2}{\Longrightarrow} oo_2 ... oo_{k-1} \overset{R^k}{\Longrightarrow} o_2$, o_1 must happen before o_2, denoted as $o_1 \Rightarrow o_2$. If neither $o_1 \Rightarrow o_2$ nor $o_2 \Rightarrow o_1$ holds, they are concurrent and hence can execute side by side in any order. This set of HB rules need to be comprehensive and precise in order for DCatch to achieve good bug detection accuracy and coverage.

[1] An operation could be a memory access, a thread creation, etc.

App	Inter-Node			Intra-Node	
	Sync. RPC	Async. Socket	Custom Protocol	Sync. Threads	Async. Events
Cassandra	-	✓	-	✓	✓
HBase	✓	-	✓	✓	✓
MapReduce	✓	-	✓	✓	✓
ZooKeeper	-	✓	-	✓	✓

Table 1. Concurrency & communication in distributed systems (Sync.: synchronous; Async.: asynchronous)

Why do we need a new model? On one hand, HB models were thoroughly studied for single-machine systems, including both multi-threaded software [32] and event-driven applications [17, 29, 36]. However, these models do not contain all causality relationships in distributed systems, and may contain causality relationships not held in distributed systems. On the other hand, distributed-system debugging tools [28, 40] proposed meta-data propagation techniques to track coarse-granularity causality relationship between *user-specified* operations. However, without a formal HB model, they are unsuitable for DCbug detection, where fine-granularity causality relationship needs to be computed among a huge number of memory accesses.

Below we present the concurrency and communication mechanisms that we learn from studying representative distributed systems (Table 1), from which we abstract HB rules.

2.1 Inter-node concurrency and communication

Every distributed system involves multiple parallel-executing nodes that communicate with each other through messages (Figure 4a). We abstract message-related HB rules, short as **Rule-M**, based on different communication patterns.

Synchronous RPC A thread in node n_1 could call an RPC function r implemented by node n_2, like step (3) in Figure 3. This thread will block until n_2 sends back the RPC result. RPC communication implies the following HB rules: making an RPC call r on n_1, denoted as *Create (r, n_1)*, happens before the beginning of the RPC execution on n_2, *Begin (r, n_2)*; the end of the RPC execution, *End (r, n_2)*, happens before the return from the RPC call r on n_1, *Join (r, n_1)*.

Rule-M$^{\text{rpc}}$: *Create* $(r, n_1) \xRightarrow{M^{\text{rpc}}}$ *Begin* (r, n_2);
End $(r, n_2) \xRightarrow{M^{\text{rpc}}}$ *Join* (r, n_1).

Asynchronous Socket A thread in node n_1 sends a message m to node n_2 through network sockets. Unlike RPC, the sender does not block and continues its execution. The sending happens before the receiving.

Rule-M$^{\text{soc}}$: *Send* $(m, n_1) \xRightarrow{M^{\text{soc}}}$ *Recv* (m, n_2).

In addition to the above two basic communication mechanisms, we also found the following common custom synchronization protocols, implemented using a combination of RPC/socket communication and intra-node computation.

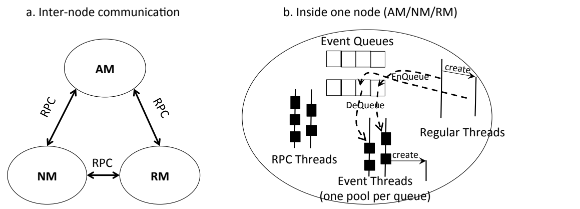


Figure 4. Concurrency and communication in MapReduce (a vertical line: a thread; a black rectangle: a RPC function or an event handler)

Custom Push-Based Synchronization Protocol Node n_1 updates a status s to a dedicated coordination node n_c, and n_c notifies all subscribed nodes, such as n_2, about this update. The update of s by n_1, *Update (s,n_1)*, happens before the notification delivered at n_2, *Pushed (s, n_2)*. For example, HBase nodes sometimes communicate through ZooKeeper: one node registers a *zknode* with a specific path in ZooKeeper; ZooKeeper will then notify this node of all changes to this *zknode* from other nodes, like steps (6) and (7) in Figure 3.

Rule-M$^{\text{push}}$: *Update* $(s, n_1) \xRightarrow{M^{\text{push}}}$ *Pushed* (s, n_2).

Note that, this rule is **not** redundant given Rule-M$^{\text{rpc}}$ and Rule-M$^{\text{soc}}$. We can decompose this rule into three chains of causality relationship: (1) *Update*$(s, n_1) \Rightarrow$ *Recv*(s, n_c); (2) *Recv*$(s, n_c) \Rightarrow$ *Send*(s, n_c); (3) *Send*$(s, n_c) \Rightarrow$ *Pushed*(s, n_2). Chain (2) is very difficult to figure out, as it involves complicated intra-node computation and synchronization in n_c, which guarantees that every node interested in s gets a notification. Even for chain (1) and (3), there is no guarantee that Rule-M$^{\text{rpc}}$ and Rule-M$^{\text{soc}}$ can figure them out, because the communication between n_1/n_2 and n_c often contains more than just one RPC or socket message.

Custom Pull-Based Synchronization Protocol Node n_2 keeps polling n_1 about status s in n_1, and does not proceed until it learns that s has been updated to a specific value. Clearly, the update of s in n_1 happens before the use of this status on n_2.

Rule-M$^{\text{pull}}$: *Update*$(s, n_1) \xRightarrow{M^{\text{pull}}}$ *Pulled*(s, n_2).

This is similar with the distributed version of the while-loop custom synchronization in single-machine systems [42, 46]. Figure 2 shows an example of this HB relationship: jMap.put (jID, task) in AM happens before the exit of the while-loop in NM.

This rule is not redundant given other rules due to complicated semantics inside n_1: traditional HB rules cannot establish the causality between s being set and s being read by an RPC function or being serialized into a socket message.

2.2 Intra-node concurrency and communication

Synchronous multi-threaded concurrency Within each node, there are multiple processes and threads, as shown in Figure 4b[2]. The rules here are about classic fork/join causality: the creation of a thread t (or process) in the parent thread, denoted as *Create(t)*, happens before the execution of t starts, *Begin (t)*. The end of t's execution, *End (t)*, happens before a successful join of t *Join (t)*.

Rule-T$^{\text{fork}}$: *Create* $(t) \overset{T^{\text{fork}}}{\Longrightarrow}$ *Begin* (t).

Rule-T$^{\text{join}}$: *End* $(t) \overset{T^{\text{join}}}{\Longrightarrow}$ *Join* (t).

Asynchronous event-driven concurrency All the systems in Table 1 conduct asynchronous event-driven processing, like steps (4)(5) in Figure 3, essentially creating concurrency inside a thread. Events could be enqueued by any thread, and then processed by pre-defined handlers in event-handling thread(s). The enqueue of an event e, *Create* (e), happens before the handler-function of e starts, denoted as *Begin* (e).

Rule-E$^{\text{enq}}$: *Create* $(e) \overset{E^{\text{enq}}}{\Longrightarrow}$ *Begin* (e).

For two events e_1 and e_2 from the same queue, the timing between their handling depends on several properties of the queue. For all the systems that we have studied, all the queues are FIFO and every queue has only one dispatching thread, one or multiple handling threads. Consequently, the handling of e_1 and e_2 is serialized when their queue is equipped with only one handling thread, and is concurrent otherwise. We refer to the former type of queues as *single-consumer queues*. All the queues in ZooKeeper and some queues in MapReduce are single-consumer queues.

Rule E$^{\text{serial}}$: *End* $(e_1) \overset{E^{\text{serial}}}{\Longrightarrow}$ *Begin* (e_2), if *Create* $(e_1) \Rightarrow$ *Create* (e_2); $e_1, e_2 \in Q$; Q is single-consumer FIFO queue.

Previous work has built HB rules for single-machine event-driven applications, particularly Android apps [17, 29, 36]. In comparison, some complicated queues (e.g., non-FIFO queues) and corresponding rules observed by previous work have not been observed in these distributed systems.

Sequential program ordering According to the classical HB model [21], the execution order within one thread is deterministic and hence has the following rule.

Rule P$^{\text{reg}}$: $o_1 \overset{P^{\text{reg}}}{\Longrightarrow} o_2$, if o_1 occurs before o_2 during the execution of a regular thread.

We need to revise this rule for threads that are involved in asynchronous computing. Specifically, for two operations inside an event/RPC/message handling thread, sequential program ordering exists between them only when they belong to the same event/RPC/message handler function.

Rule P$^{\text{nreg}}$: $o_1 \overset{P^{\text{nreg}}}{\Longrightarrow} o_2$, if o_1 occurs before o_2 during the execution of an event handler, a message handler, or an RPC function.

2.3 Summary

The above **MTEP** rules constitute the DCatch HB model. Our evaluation will show that every rule is crucial to the accuracy and coverage of DCbug detection (Section 7.4). For the real-world example demonstrated in Figure 3, we can now infer $W \Rightarrow R$, because of the following chains of happens-before relationship: $W \overset{P^{\text{reg}}}{\Longrightarrow}$ *Create (t)* $\overset{T^{\text{fork}}}{\Longrightarrow}$ *Begin (t)* $\overset{P^{\text{reg}}}{\Longrightarrow}$ *Create (OpenRegion, HMaster)* $\overset{M^{\text{rpc}}}{\Longrightarrow}$ *Begin (OpenRegion, HRS)* $\overset{P^{\text{nreg}}}{\Longrightarrow}$*Create (e)* $\overset{E^{\text{enq}}}{\Longrightarrow}$*Begin (e)* $\overset{P^{\text{nreg}}}{\Longrightarrow}$*Update (RS...OPENED, HRS)* $\overset{M^{\text{push}}}{\Longrightarrow}$*Pushed (RS...OPENED, HMaster)* $\overset{P^{\text{nreg}}}{\Longrightarrow} R$.

Note that, our model is not the only viable HB model for distributed systems. Our model abstracts away some low-level details in RPC and event libraries. For example, incoming RPC calls are first put into queue(s) before assigned to RPC threads, but our Rule-M$^{\text{rpc}}$ abstracts away these queues inside RPC library; between the enqueue of an event and the beginning of the event handling, a dedicated thread would conduct event dispatching, which is also abstracted away in our Rule-E$^{\text{enq}}$.

Our model also intentionally ignores certain causality relationships that do not affect our DCbug detection. For example, our model does not consider condition-variable notify-and-wait causality relationship, because it is almost never used in the inter-node communication and computation part of our studied distributed systems; we do not consider lock synchronization in this model, because lock provides mutual exclusions not strict ordering.

Our model could also miss some custom synchronization protocols in distributed systems.

Next few sections will describe the design of the four components of DCatch based on the model defined above.

3. DCatch tracing and trace analysis

Given our HB model, we began building the DCatch tool. As first steps, we need to (1) trace the necessary operations and (2) build the HB graph and perform analysis on top. Below we describe how these work and how we address tracing and analysis challenges such as reducing memory access traces and applying the MTEP rules correctly.

3.1 DCatch Tracing

DCatch produces a trace file for every thread of a target distributed system at run time. These traces will then allow trace analyzer to identify DCbug candidates. The detailed implementation is based on WALA, a static Java bytecode analysis framework, and Javassist, a dynamic Java bytecode transformation framework; more details are in Section 6.

3.1.1 Which operations to trace?

Memory-access tracing Naively, we want to record all accesses to program variables that could potentially be shared among threads or event handlers. However, this exhaustive approach would lead to huge traces that are expensive or even cannot be processed for many real-world distributed system workloads as we will see in Section 7.4.

Fortunately, such excessive logging is unnecessary for DCbug detection. DCbugs are triggered by inter-node in-

[2] MapReduce contains multiple processes in one node not shown in figure.

	M-Rule	T-Rule	E-Rule	P-Rule
Creat (t), Join (t)		✓		
Begin (t), End (t)		✓		
Begin (e), End (e)			✓	✓
Create (e)			✓	
Begin (r, n_2), End (r, n_2)	✓			✓
Create (r, n_1), Join (r, n_1)	✓			
Send (m, n_1)	✓			
Recv (m, n_2)	✓			✓
Update (s, n_1)	✓			
Pushed (s, n_2)	✓			✓
Pull (s, n_2)	✓			

Table 2. HB-related tracing (symbols defined in Section 2)

teraction, with the root-cause memory accesses in code regions related to inter-node communication and corresponding computation, not everywhere in the software.

Following this design principle, DCatch traces all accesses to heap objects and static variables in the following three types of functions and their callees: (1) RPC functions; (2) functions that conduct socket operations; and (3) event-handler functions. The third type is considered because they conduct many pre- and post-processing of socket sending/receiving and RPC calls.

HB-related operation tracing DCatch traces operations that allow its trace analysis to apply the **MTEP** rules, as shown in Table 2. DCatch automatically identifies these operations at run time using the Javassist infrastructure. The implementation details are in Section 6.

For push-based synchronization, the current prototype of DCatch focuses on the synchronization service provided by ZooKeeper, as discussed in Section 2.1. DCatch traces ZooKeeper APIs ZooKeeper::create, ZooKeeper::delete, and ZooKeeper::setData as *Update* operations, and ZooKeeper Watcher events with event types NoteCreated, NodeDeleted, and NodeDataChanged as *Push* operations. The parameters, event types, and timestamps help DCatch trace analysis to group corresponding *Update* and *Push* together. For pull-based synchronization, the *Update* and *Pull* operations involve memory accesses, RPC calls, and loops, which are already traced. We will explain how to put them together to construct pull-based HB relationship in Section 3.2.1.

Other tracing DCatch does not need to trace lock and unlock operations to *detect* DCbugs, because lock and unlock operations are not part of the DCatch HB model. However, as we will see in Section 5.2, DCatch needs to know about lock/unlock operations to *trigger* some DCbug candidates. Such information sometimes can help avoid hangs when DCatch tries to manipulate the timing and *trigger* a DCbug candidate. Therefore, DCatch also traces lock and unlock operations, including both implicit lock operations (i.e., synchronized methods and synchronized statements) and explicit lock operations.

3.1.2 What to record for each traced operation?

Each trace record contains three pieces of information: (1) type of the recorded operation; (2) callstack of the recorded operation; and (3) ID. The first two are straightforward. The IDs help DCatch trace analyzer to find related trace records.

For a memory access, ID uniquely identifies the accessed variable or object. The ID of an object field is the field-offset and the object hashcode. The ID of a static variable is the variable name and its corresponding namespace.

For HB-related operations, the IDs will allow DCatch trace analysis to correctly apply HB rules. For every thread- or event- related operation, the ID is the object hashcode of the corresponding thread or event object. For each RPC-related and socket-related operation, DCatch tags each RPC call and each socket message with a random number generated at run time (details in Section 6).

For lock/unlock operations, the IDs uniquely identify the lock objects, allowing DCatch's triggering module to identify all lock critical sections and perturb the timing at appropriate places (details in Section 5.2).

3.2 DCatch trace analysis

DCatch trace analyzer identifies every pair of memory accesses $(s,\ t)$, where s and t access the same variable with at least one write and are concurrent with each other (i.e., no HB-relationship between them), and considers $(s,\ t)$ as a DCbug candidate.

3.2.1 HB-graph construction

An HB graph is a DAG graph. Every vertex v represents an operation $o(v)$ recorded in DCatch trace, including both memory accesses and HB-related operations. The edges in the graph are arranged in a way that v_1 can reach v_2 if and only if $o(v_1)$ happens before $o(v_2)$.

To build such a graph, DCatch first goes through all trace files collected from all threads of all processes in all nodes, and makes every record a vertex in the graph.

Next, DCatch adds edges following our MTEP rules. We discuss how to apply **Rule $\mathbf{E}^{\text{serial}}$** and **Rule $\mathbf{M}^{\text{pull}}$** below. We omit the details of applying other rules, as they are straightforward and can be applied in any order — the ID of each trace record allows DCatch to easily group related operations.

DCatch applies **Rule $\mathbf{E}^{\text{serial}}$** as the last HB rule. For every thread that handles a single-consumer event queue, DCatch checks every pair of *End* (e_i) and *Begin* (e_j) recorded in its trace, and adds an edge from the former to the latter, if DCatch finds *Create* $(e_i) \Rightarrow$ *Create* (e_j) based on those HB edges already added so far. DCatch repeats this step until reaching a fixed point.

Applying **Rule $\mathbf{M}^{\text{pull}}$** requires program analysis. The algorithm here is inspired by how loop-based custom synchronization is handled in LCbug detection [42, 46]. For every pair of conflicting concurrent read and write $\{r,\ w\}$, we consider r to be potentially part of a pull-based synchronization protocol if (1) r is executed inside an RPC function; (2) the return value of of this RPC function depends on r; (3) in another node that requests this RPC, the return value of this RPC is part of the exit condition of a loop l. We will then run the targeted software again, tracing only such rs and all writes that touch the same object based on the original trace. The new trace will tell us which write $w*$ provides value for the last instance of r before l exits. If $w*$ and r are from different threads, we will then conclude that $w*$ in one node happens before the exit of the remote loop l in another node. Due to space constraints, we omit the analysis details here. This part of the analysis is done together with intra-node while-loop synchronization analysis. Although requiring running the software for a second time, it incurs little tracing or trace analysis overhead, because it focuses on loop-related memory accesses.

3.2.2 DCbug candidate report

The HB graph is huge, containing thousands to millions of vertices in our experiments. Naively computing and comparing the vector-timestamps of every pair of vertices would be too slow. Note that each vector time-stamp will have a huge number of dimensions, with each event handler and RPC function contributing one dimension.

To speed up this analysis, DCatch uses the algorithm proposed by previous asynchronous race detection work [36]. The algorithm there computes a reachable set for every vertex in HB graph, represented by a bit array, and then turns HB-relationship checking into a constant-time array lookup.

4. Static pruning

Not all DCbug candidates reported by trace analysis can cause failures. This is particularly true in distributed systems, which inherently contain more redundancy and failure tolerance than single-machine systems. The high-level idea of pruning false positives by estimating failure impacts has been used by previous LCbug detection tools [49, 50]. However, previous work only analyzes intra-procedural failure impacts. Thus, the challenge is to conduct inter-procedural *and* inter-node impact analysis to better suit the failure-propagation nature of DCbugs in distributed systems.

To avoid excessive false positives, we first configure DCatch to treat certain instructions in software as *failure instructions*, which represent the (potential) occurrence of severe failures. Then, given a bug candidate (s, t), DCatch statically analyzes related Java bytecode of the target system to see if s or t may have local (i.e., within one node) or distributed (i.e., beyond one node) impact towards the execution of any failure instruction identified above.

4.1 Identifying failure instructions

The current prototype of DCatch considers the following failures and identifies *failure instructions* accordingly: (1) system aborts and exits, whose corresponding failure instructions are invocations of abort and exit functions (e.g., $\mathrm{System.exit}$); (2) severe errors that are printed out, whose corresponding failure instructions are invocations of $\mathrm{Log::fatal}$ and $\mathrm{Log::error}$ functions in studied systems; (3) throwing uncatchable exceptions, such as $\mathrm{RuntimeException}$; (4) infinite loops, where we consider every loop-exit instruction as a potential failure instruction. Finally, if a failure instruction is inside a catch block, we also consider the corresponding exception throw instruction, if available, as a failure instruction. This list is configurable, allowing future DCatch extension to detect DCbugs with different failures.

4.2 Impact estimation

For a DCbug candidate (s, t), if DCatch fails to find any failure impact for s and t through the analysis described below, this DCbug candidate will be pruned out from the DCatch bug list. All the implementation below is done in WALA code analysis framework, leveraging WALA APIs that build program dependency graphs.

Local impact analysis We conduct both intra-procedural and inter-procedural analysis for local impact analysis. Given a memory-access statement s located in method M, we first check whether any failure instruction in M has control- or data- dependence on s. We apply similar checking for t.

We then check whether s could affect failure instructions inside the callers of M through either the return value of M or heap/global objects. For the latter, DCatch only applies the analysis to one-level caller of M, not further up the call chain for accuracy concerns. Note that, since DCatch tracer and trace analysis report call-stack information, our inter-procedural analysis follows the reported call-stack of s. Finally, we check whether s could affect failure sites in the callee functions of M through either function-call parameters or heap/global variables. This analysis is also only applied to the one-level callee of M. We skip our algorithm details due to space constraints.

Distributed impact analysis As shown in Figure 2, an access in one node could lead to a failure in a different node. Therefore, DCatch also analyzes RPC functions to understand the remote impact of a memory access.

Specifically, if we find an RPC function R along the callstack of the memory access s, we check whether the return value of R depends on s. If so, we then locate the function M_r on a different node that invokes the RPC call R. Inside M_r, we check whether any failure instruction depends on the return value of R. Note that locating M_r is straightforward given the HB chains already established by DCatch trace analysis.

DCatch does not analyze inter-node impact through sockets, as socket communication is not as structured as RPCs.

5. DCBug triggering and validation

A DCatch bug report (s, t) still may not be harmful for two reasons. First, s and t may not be truly concurrent with each other due to custom synchronization unidentified by DCatch. Second, the concurrent execution of s and t may not lead to any failures, as the impact analysis conducted in Section 4 only provides a static estimation. Furthermore, even for those truly harmful DCbug candidates, triggering them could be very challenging in distributed systems.

To address this, we do not stop with just reporting potential DCbugs, but rather we also build this last component of DCatch to help assess DCbug reports and reliably expose truly harmful DCbugs, hence an end-to-end analysis-to-testing tool. This phase includes two parts: (1) an infrastructure that enables easy timing manipulation in distributed systems; and (2) an analysis tool that suggests how to use the infrastructure to trigger a DCbug candidate. These two features are unique to triggering DCbugs.

5.1 Enable timing manipulation

Naively, we could perturb the execution timing by inserting sleep into the program, like how LCbugs are triggered in some previous work [35]. However, this naive approach does not work for complicated bugs in complicated systems, because it is hard to know how long the sleep needs to be. More sophisticated LCbug exposing approach [31, 38] runs the whole program in one core and controls the timing through thread scheduler. This approach does not work for DCbugs, which may require manipulating the timing among operations from different nodes, in real-world large distributed systems, which are impractical to run on one core.

Our infrastructure includes two components: client-side APIs for sending coordination-request messages and a message-controller server (we refer to the distributed system under testing as client here).

Imagine we are given a pair of operations A and B, and we want to explore executing A right before B and also B right before A. We will simply put a _request API call before A and a _confirm API call right after A, and the same for B. At run time, the _request API will send a message to the controller server to ask for the permission to continue execution. At the controller side, it will wait for the request-message to arrive from both parties, and then grant the permission to one party, wait for the confirm-message sent by the _confirm API, and finally grant the permission for the remaining party. The controller will keep a record of what ordering has been explored and will re-start the system several times, until all ordering permutations among all the request parties (just two in this example) are explored.

5.2 Design timing-manipulation strategy

With the above infrastructure, the remaining question is where to put the _request and _confirm APIs given a DCbug report (s, t). The _confirm APIs can be simply inserted right after the heap access in the bug report. Therefore, our discussion below focuses on the placement of _request APIs.

The naive solution is to put _request right before s and t. However, this naive approach may lead to hangs or too many _request messages sent to the controller server due to the huge number of dynamic instances of s or t. DCatch provides the following analysis to help solve this problem, both are unique to triggering DCbugs.

First, DCatch warns about potential hangs caused by poor placements of _request in the following three cases and suggests non-hang placements. (1) If s and t are both inside event handlers and their event handlers correspond to a single-consumer queue, DCatch warns about hangs and suggests putting _request in corresponding event enqueue functions. (2) If s and t are both inside RPC handlers and their RPC functions are executed by the same handling thread in the same node, DCatch suggests putting _request in corresponding RPC callers. (3) If s and t are inside critical sections guarded by the same lock, DCatch suggests putting _request right before the corresponding critical sections. DCatch gets the critical section information based on lock-related records in its trace, as discussed in Section 3.1.

Second, DCatch warns about large number of dynamic instances of s and t and suggest better placements. The DCBug report will contain call-stacks for s and t. When DCatch checks the run-time trace and finds a large number of dynamic instances of the corresponding call-stack for s (same for t), DCatch will check its happens-before graph to find an operation o in a different node that causes s, and checks whether o is a better place for _request. This analysis is very effective: many event handlers and RPC functions are always executed under the same call stack, and hence could make bug triggering very complicated without this support from DCatch.

6. Implementation

DCatch is implemented using WALA v1.3.5 and Javassist v3.20.0 for a total of 12 KLOC. Below are more details.

HB-related operation tracing DCatch traces HB-related operations using Javassist, a dynamic Java bytecode rewriting tool, which allows us to analyze and instrument Java bytecode whenever a class is loaded.

All thread-related operations can be easily identified following the java.lang.Thread interface. Event handling is implemented using org.apache.hadoop.yarn.event.EventHandler and org.apache.hadoop.hbase.executor.EventHandler interface in Hadoop and HBase. The prototype of an handler function is EventHandler::handle (Event e). Cassandra and ZooKeeper use their own event interfaces. The way handler functions are implemented and invoked are similar as that in Hadoop/HBase.

For RPC, HBase and early versions of Hadoop share the same RPC library interface, VersionedProtocol. All methods declared under classes instantiated from this interface are RPC functions, and hence can be easily identified. Later versions of Hadoop use a slightly different interface, ProtoBase, but the way to identify its RPC functions is similar.

For socket, Cassandra has a superclass IVerbHandler to handle socket communication and every message sending is conducted by IVerbHandler::sendOneWay (Message, EndPoint). DCatch can easily identify all such function calls, as well as the message object. ZooKeeper uses a super-class Record for all socket messages. DCatch identifies socket sending and receiving based on how Record objects are used.

Memory access tracing DCatch first uses WALA, a static Java bytecode analysis framework, to statically analyze the target software, identifies all RPC/socket/event related functions, and stores the result. DCatch then uses Javassist to insert tracing functions before every heap access (getfield/putfield instruction) or static variable access (getstatic/putstatic instruction) in functions identified above.

Tagging RPC DCatch statically transforms the target software, adding one extra parameter for every RPC function and one extra field in socket-message object, and inserting the code to generate a random value for each such parameter/field at the invocation of every RPC/socket-sending function. DCatch tracing module will record this random number at both the sending side and the receiving side, allowing trace analysis to pair message sending and receiving together.

Portability of DCatch As described above, applying DCatch to a distributed software project would require the following information about that software: (1) what is the RPC interface; (2) what are socket messaging APIs; (3) what are event enqueue/dequeue/handler APIs; (4) whether the event queues are FIFO and whether they have one or multiple handler threads.

In our experience, providing the above specifications is straightforward and reasonably easy, because we only need to identify a small number of (RPC/event/socket) interfaces or prototypes, instead of a large number of instance functions. We also believe that the above specifications are necessary for accurate DCbug detection in existing distributed systems, just like specifying pthread functions for LCbug detection and specifying event related APIs for asynchronous-race detection.

7. Evaluation

7.1 Methodology

Benchmarks We evaluate DCatch on seven timing-related problems reported by real-world users in four widely used open-source distributed systems: Cassandra distributed key-value stores (CA); HBase distributed key-value stores (HB); Hadoop MapReduce distributed computing framework (MR); ZooKeeper distributed synchronization service (ZK). These systems range from about 61 thousand lines of code to more than three million lines of code, as shown in Table 3.

BugID	LoC	Workload	Symptom	Error	Root
CA-1011	61K	startup	Data backup failure	DE	AV
HB-4539	188K	split table & alter table	System Master Crash	DE	OV
HB-4729	213K	enable table & expire server	System Master Crash	DE	AV
MR-3274	1,266K	startup + wordcount	Hang	DH	OV
MR-4637	1,388K	startup + wordcount	Job Master Crash	LE	OV
ZK-1144	102K	startup	Service unavailable	LH	OV
ZK-1270	110K	startup	Service unavailable	LH	OV

Table 3. Benchmark bugs and applications.

We obtain these benchmarks from TaxDC benchmark suite [24]. They are all triggered by untimely communication across nodes. As shown in Table 3, they cover all common types of failure symptoms: job-master node crash, system-master node crash, hang, etc. They cover different patterns of errors: local explicit error (LE), local hang (LH), distributed explicit error (DE), distributed hang (DH). Here, local means on the same machine as the root-cause memory accesses; distributed means on a different machine from the root-cause accesses. They also cover different root causes: order violations (OV) and atomicity violations (AV).

Experiment settings We use failure-triggering workloads described in the original user reports, as shown in Table 3. They are actually common workloads: system startups in Cassandra and ZooKeeper; alter a table and then split it in HBase; enable a table and then crash a region-server in HBase; run WordCount (or any MapReduce job) and kill the job before it finishes in MapReduce. Note that, due to the non-determinism of DCbugs, failures rarely occur under these workload. DCatch detects DCbugs by monitoring **correct** runs of these workload.

We run each node of a distributed system in one virtual machine, and run all VMs in one physical machine (M1), except for HB-4539, which requires two physical machines (M1 & M2). Both machines use Ubuntu 14.04 and JVM v1.7. M1 has Intel® Xeon® CPU E5-2620 and 64GB of RAM. M2 has Intel® Core^{TM}i7-3770 and 8GB of RAM. We connect M1 and M2 with Ethernet cable. All trace analysis and static pruning are on M1. After the static pruning in Section 4, all the remaining bug candidates are considered **DCatch bug reports**. We will then try to trigger each reported bug leveraging the DCatch triggering module. Note that, the triggering result does **not** change the count of DCatch bug reports.

Evaluation metrics We will evaluate the following aspects of DCatch: the coverage and accuracy of bug detection, and the overhead of bug detection, including run-time overhead, off-line analysis time, and log size. All the performance numbers are based on an average of 5 runs. We will also compare DCatch with a few alternative designs.

We will put a DCatch bug report (s, t) into one of three categories: if s and t are not concurrent with each other, it is a *serial* report (i.e., not concurrent); if s and t are concurrent with each other, but their concurrent execution does not lead

BugID	Detected?	#Static Ins. Pair			#CallStack Pair		
		Bug	Benign	Serial	**Bug**	Benign	Serial
CA-1011	✓	$\mathbf{3}_1$	0	0	$\mathbf{5}_1$	2	0
HB-4539	✓	$\mathbf{3}_3$	0	1	$\mathbf{3}_3$	0	1
HB-4729	✓	$\mathbf{4}_4$	1	0	$\mathbf{5}_5$	5	0
MR-3274	✓	$\mathbf{2}_1$	0	4	$\mathbf{2}_1$	0	9
MR-4637	✓	$\mathbf{1}_1$	2	4	$\mathbf{1}_1$	3	9
ZK-1144	✓	$\mathbf{5}_1$	1	1	$\mathbf{5}_1$	1	1
ZK-1270	✓	$\mathbf{6}_1$	2	0	$\mathbf{6}_1$	2	0
Total*		$\mathbf{20}_{12}$	5	7	$\mathbf{23}_{13}$	12	12

Table 4. DCatch bug detection results (*: Total is smaller than sum of all rows, as we do **not** double count same bug reported by two benchmarks. Subscript denotes bug reports related to the known bug listed in Column-1. Definitions of columns are in Section 7.1.)

to failures, it is a *benign* bug; if their concurrent execution leads to failures, it is a true *bug*.

We report DCbug counts by the unique number of static instruction pairs and the unique number of callstack pairs as shown in Table 4. Since these two numbers do not differ much, we use static-instruction count by default unless otherwise specified.

7.2 Bug detection results

Overall, DCatch has successfully detected DCbugs for all benchmarks while monitoring *correct* execution of these applications, as shown by the ✓ in Table 4. In addition, DCatch found a few truly harmful DCbugs we were unaware of and outside the TaxDC suite [24]. DCatch is also accurate: only about one third of all the 32 DCatch bug reports are false positives based on static count.

Harmful bug reports DCatch has found root-cause DCbugs for every benchmark. In some cases, DCatch found multiple root-cause DCbugs for one benchmark. For example, in HB-4729, users report that "clash between region unassign and splitting kills the master". DCatch found that one thread t_1 could delete a zknode concurrently with another thread t_2 reads this zknode and deletes this zknode. Consequently, multiple DCbugs are reported here between delete and reads, and between delete and delete. They are all truly harmful bugs: any one of these zknode operations in t_2 would fail and cause HMaster to crash, if the delete from t_1 executes right before it.

DCatch also found a few harmful DCbugs, 8 in static count and 10 in callstack count, that go beyond the 7 benchmarks. We were unaware of these bugs, and they are not part of the TaxDC bug suite. We have triggered all of them and observed their harmful impact, such as node crashes and unavailable services, through DCatch triggering module. We have carefully checked the change log of each software project, and found that two among these 8 DCbugs have never been discovered or patched and the remaining have already been patched by developers in later versions.

BugID	#Static Ins. Pair			#Callstack Pair		
	TA	TA+SP	TA+SP+LP	TA	TA+SP	TA+SP+LP
CA-1011	46	4	3	175	9	7
HB-4539	24	4	4	57	5	4
HB-4729	52	6	5	219	12	10
MR-3274	53	8	6	553	18	11
MR-4637	61	8	7	568	21	13
ZK-1144	29	8	7	52	8	7
ZK-1270	25	10	8	25	10	8

Table 5. # of DCbugs reported by trace analysis (TA) alone, then plus static pruning (SP), then plus loop-based synchronization analysis (LP), which becomes DCatch.

Benign bug reports DCatch only reported few benign DCbugs, 5 out of 32 across all benchmarks, benefitting from its static pruning module. In Cassandra, DCatch reports some DCbugs that can indeed cause inconsistent metadata across nodes. However, this inconsistency will soon get resolved by the next gossip message. Therefore, they are benign. Other benign reports are similar. Note that, for CA-1011, the benign report count is 0 in static count but 2 in callstack count, because the two benign reports share the same static identities with some truly harmful bug reports.

Serial bug reports DCatch HB model and HB analysis did well in identifying concurrent memory accesses. For only 7 out of 32 DCbug reports, DCatch mistakenly reports two HB-ordered memory accesses as concurrent. Some of them are caused by unidentified RPC functions, which do not follow the regular prototype and hence are missed by our static analysis. Some of them are caused by custom synchronization related to address transfer [48]. The remaining are caused by distributed custom synchronization. For example, ZK has a function waitForEpoch, essentially a distributed barrier — accesses before waitForEpoch in n_1 happens before accesses after corresponding waitForEpoch in n_2. The implementation of waitForEpoch is complicated and cannot be inferred by existing HB rules. Single-machine custom synchronization is an important research topic in LCbug detection [42, 46]. DCatch is just a starting point for research on distributed custom synchronization.

DCatch false-positive pruning As shown in Table 5, our static pruning pruned out a big portion of DCbug candidates reported by DCatch trace analysis: less than 10% of DCbug candidates (callstack count) remain after the static pruning for CA, HB, and MR benchmarks.

To evaluate the quality of static pruning, we randomly sampled and checked 35 DCbug candidates that have been pruned out, 5 from each benchmark. We found that all of them are indeed false positives. A few of them would lead to exceptions, but the exceptions are well handled with only warning or debugging messages printed out through LOG.warn or LOG.debug.

Of course, our static pruning could prune out truly harmful bugs. However, given the huge number of DCbug candidates reported by DCatch trace analysis, DCatch static pruning is valuable for prioritizing the bug detection focus.

Finally, our loop-based synchronization analysis is effective (Section 3.2.1). This analysis discovers both local while-loop custom synchronization and distributed pull-based custom synchronization. It pruned out false positives even after the intensive static pruning for all benchmarks, as shown in Table 5.

Triggering Overall, DCatch triggering module has been very useful for us to trigger DCbugs and prune out false positives. As shown in Table 4, among the 47 DCatch bug reports with unique call stacks, the triggering module is able to automatically confirm 35 of them to be true races, with 23 of them causing severe failures, and the remaining 12 to be false positives in DCatch race detection.

The analysis conducted by DCatch about how to avoid hangs (challenge-1) and avoid large numbers of dynamic _requests (challenge-2) is crucial to trigger many DCatch bug reports. In fact, the naive approach that inserts _request just before the racing heap accesses failed to confirm 23 DCatch bug reports to be true races, out of the total 35 true races, exactly due to these two challenges. DCatch handles the challenge-1 by putting _requests outside critical sections (17 cases) or outside event/RPC handlers (6 cases), and handles the challenge-2 by moving the _requests along the happens-before graph into nodes different from the original race instructions (2 cases), exactly like what described in Section 5.2. In two cases, to avoid hangs, DCatch first move _request from inside RPC handlers into RPC callers and then move _request to be right outside the critical sections that enclose corresponding RPC callers.

Of course, DCatch triggering module is not perfect. We expect two remaining challenges that it could encounter. First, DCatch cannot guarantee to find a location along the HB chains with few dynamic instances. Automated DCbugs triggering would be challenging in these cases, if failures only happen at a specific dynamic instance. The current prototype of DCatch focuses on the first dynamic instance of every racing instruction. This strategy allows DCatch to trigger the desired executing order among race instructions with 100% frequency for 33 true races in DCatch bug reports and with about 50% frequency for the remaining 2 true races. Second, DCatch does not record all non-deterministic environmental events and hence its triggering module may fail to observe a race instruction whose execution depends on unrecorded non-deterministic events.

False negative discussion DCatch is definitely not a panacea. DCatch could miss DCbugs for several reasons. First, given how its static pruning is configured, the current prototype of DCatch only reports DCbugs that lead to explicit failures, as discussed in Section 4.1. True DCbugs that lead to severe but silent failures would be missed. This problem could be addressed by skipping the static pruning step, and simply applying the triggering module for all DCbug candidates. This could be an option if the testing budget allows. Second, DCatch selectively monitors only memory accesses related to inter-node communication and corresponding computation. This strategy is crucial for the scalability of DCatch, as we will see soon in Section 7.4. However, there could be DCbugs that are between communication-related memory accesses and communication-unrelated accesses. These bugs would be missed by DCatch. Fortunately, they are very rare in real world based on our study. Third, DCatch may not process extremely large traces. The scalability bottleneck of DCatch, when facing huge traces, is its trace analysis. It currently takes about 4G memory for the three largest traces in our benchmarks (HB-4729, MR-3274, and MR-4637). DCatch will need to chunk the traces and conduct detection within each chunk, an approach used by previous LCbug detection tools.

BugID	Base	Tracing	Trace Analysis	Static Pruning	Trace Size
CA-1011	6.6s	13.0s	15.9s	324s	7.7MB
HB-4539	1.1s	3.8s	11.9s	87s	4.9MB
HB-4729	3.5s	16.1s	36.8s	278s	19MB
MR-3274	21.2s	94.4s	62.2s	341s	22MB
MR-4637	11.7s	36.4s	51.5s	356s	18MB
ZK-1144	0.8s	3.6s	4.8s	25s	1.9MB
ZK-1270	0.2s	1.1s	4.5s	15s	1.3MB

Table 6. DCatch Performance Results (Base is the execution time of each benchmark without DCatch).

7.3 Performance results

Run-time and off-line analysis time As shown in Table 6, DCatch performance is reasonable for in-house testing. DCatch tracing causes 1.9X – 5.5X slowdowns across all benchmarks. Furthermore, we found that up to 60% of the tracing time is actually spent in dynamic analysis and code transformation in Javassist. If we use a static instrumentation tool, the tracing performance could be largely improved. The trace analysis time is about 2–10 times of the baseline execution time. Fortunately, it scales well, roughly linearly, with the trace size: taking about 2–3 second to process every 1MB of trace.

The static pruning phase takes 15 seconds to about 6 minutes for each benchmark. It is the most time consuming phase in DCatch as shown in the table. 20% – 89% of this analysis time is spent for WALA to build the Program Dependency Graph (PDG). Therefore, the pruning time would be greatly reduced, if future work can pre-compute the whole program PDG, store it to file, and load it on demand.

The time consumed by loop-based synchronization analysis is negligible comparing with tracing, trace analysis, and static pruning, and hence is not included in Table 6.

BugID	Total	Mem	RPC / Socket	Event	Thread	Lock
CA-1011	19,984	17,722	0 / 196	0	634	1432
HB-4539	3,907	3,233	260 / 0	21	89	304
HB-4729	11,297	9,694	449 / 0	18	144	992
MR-3274	31,526	23,528	752 / 0	3,540	1,390	2316
MR-4637	24,437	17,201	406 / 0	2,996	1,780	2054
ZK-1144	3,820	3,303	0 / 120	0	79	318
ZK-1270	5,367	4,227	0 / 389	0	329	422

Table 7. Break downs of # of major types of trace records

BugID	Trace Size	Tracing Time	TraceAnalysis Time
CA-1011	77MB	15.9s	Out of Memory
HB-4539	26MB	10.2s	64.5s
HB-4729	60MB	49.9s	Out of Memory
MR-3274	839MB	215.3s	Out of Memory
MR-4637	639MB	137.8s	Out of Memory
ZK-1144	6.9MB	5.7s	6.5s
ZK-1270	25MB	4.4s	232.7s

Table 8. Full Memory Tracing Results.

Tracing Details DCatch produces 1.2–21MB of traces for these benchmarks. These traces could have been much larger if DCatch did not selectively trace memory accesses, as we will see in Table 8. As shown in Table 7, these traces mostly contain memory access information. There are also a good number of RPC, socket, event, and thread related records in DCatch traces. MapReduce benchmarks particularly have many event and thread related records, because MapReduce heavily uses event-driven computation. There are many event-handling threads and many event handlers further spawn threads. On the other hand, our workload did not touch the event-driven computation part of Cassandra and Zookeeper, consequently their traces do not contain event operations.

7.4 Comparison with alternative designs

Unselective memory-access logging DCatch tracing only selectively traces memory accesses related to inter-node communication and computation. This design choice is crucial in making DCatch scale to real-world systems. As shown in Table 8, full memory-access tracing will increase the trace size by up to 40 times. More importantly, for 4 out of the 7 benchmarks, trace analysis will run out of JVM memory (50GB of RAM) and cannot finish.

Alternative HB design DCatch HB model contains many rules. We want to check whether they have all taken effects in DCbug detection, particularly those rules that do not exist in traditional multi-threaded programs. Therefore, we evaluated how many extra false positives and false negatives are reported by DCatch trace analysis when it ignores event, RPC, socket, and push-synchronization operations in traces, respectively, as shown in Table 9. Note that, the traces are the same as those used to produce results in Table 4, except that some trace records are ignored by analyzer.

BugID	#Static Ins. Pair			#Callstack Pair		
	Event	RPC_Soc	Push	Event	RPC_Soc	Push
CA-1011	-	-	-	-	-	-
HB-4539	-	-3/+35	-3/+35	-	-12/+115	-11/+110
HB-4729	-	-7/+37	-9/+36	-	-23/+109	-24/+106
MR-3274	-46/+4	-5/+16	-	-349/+ 8	-20/+ 18	-
MR-4637	-51/+4	-4/+17	-	-369/+11	-24/+ 20	-
ZK-1144	-	-	-	-	-	-
ZK-1270	-	-	-	-	-	-

Table 9. False negatives (before '/') and false positives (after '/') of ignoring certain HB-related operations (RPC and socket are in one column as they are never used together in one benchmark).

Overall, modeling these HB-related operations are *all* very useful. Excluding any one type of them would lead to a good number of false positives and false negatives for multiple benchmarks, as shown in Table 9.

The false positives are easy to understand. Without these operations, corresponding HB relationships, related to Rule-E^{enq}, Rule-M^{RPC}, Rule-M^{soc}, and Rule-M^{push}, would be missed by trace analysis. Consequently, some memory access pairs would be mistakenly judged as concurrent.

The false negatives are all related to Rule-P^{nreg}. For example, when event-handler *Begin*s and *End*s are not traced, DCatch trace analysis would conclude that all memory accesses from the same event-handling thread are HB ordered. Consequently, DCatch would miss DCbugs caused by conflicting memory accesses from concurrent event handlers. The same applies to false negatives caused by not tracking RPC/socket and Push-based synchronization operations.

Finally, CA-1011 and the two ZK benchmarks did not encounter extra false positives or negatives in static/callstack counts[3] due to lucky "*two wrongs make a right*": ignoring socket-related operations misses some true HB relationships and also mistakenly establishes some non-existing HB relationships. Imagine node n_1 sends a message m_1 to node n_2, and n_2 sends m_2 back in response. Tracing socket operations or not would both reach the conclusion that send m_1 happens before receive m_2 on n_1, through different reasoning. Tracing socket operations provide correct HB relationships: *Send* $(m_1, n_1) \overset{M^{soc}}{\Longrightarrow}$ *Recv* $(m_1, n_2) \overset{P^{nreg}}{\Longrightarrow}$ *Send* $(m_2, n_2) \overset{M^{soc}}{\Longrightarrow}$ *Recv* (m_2, n_1). Not tracing socket would mistakenly apply Rule-P^{reg} to message-handling threads, and do the wrong reasoning: *Send* $(m_1, n_1) \overset{P^{reg}}{\Longrightarrow}$ *Recv* (m_2, n_1). In short, tracing socket operations is still useful in providing accurate HB relationships.

[3] CA-1011 did encounter 8% more false positives in raw counts – each pair of callstacks may have many dynamic instances.

8. Related Work

For combating DCbugs in real distributed systems *implementations*, to the best of our knowledge there are mainly two approaches: software model checking and verification.

Model checking Distributed system model checkers (or "dmck" in short) have many variance proposed in literature recently (*e.g.*, Demeter [14], MaceMC [20], SAMC [23], Dbug [41], and MoDist [47]). Dmck is an implementation-level model checker, that is, it works by intercepting non-deterministic distributed events and permuting their ordering. Distributed events include message arrivals, fault/reboot timings, as well as local computation. Although powerful, dmck has one major weakness: the more events included, the larger the state space to be explored. It can take hours or days to explore some of the state space [23].

DCatch like many other run-time bug detection tools does not provide any guarantee of finding all possible bugs in software and does not guarantee free of false positives, but is much faster than dmck. Furthermore, DCatch, like many other run-time bug detection tools, are built to focus on bugs with certain type of root causes. Consequently, once detecting a bug, DCatch can pinpoint the bug root cause, which could be valuable in bug fixing. This root-cause pinpointing is not a goal of dmck.

Verification. The other set of work is based on verification (*e.g.*, IronFleet [15], Verdi [45]) which uses proving frameworks such as Coq [1] and TLA [22] to verify distributed system implementations. While verification/formal methods is a stronger solution (no false positives and negatives) than bug detection tools, such methods come with a cost: long proof. Just for the basic read and write protocols, the length of the proofs can reach thousands of lines of code, potentially larger than the protocol implementation.

A similar but slightly different approach is to build distributed systems with verifiable language (*e.g.*, P# [7]). This approach is still an ongoing development. Today's deployed systems are still mostly written in imperative low-level languages such as C++ or Java as performance is crucial, especially in core-service distributed systems.

LCbug and DCbug detection LCbug detection has been studied for decades. Many bug detectors have been proposed to identify data races [19, 32, 37, 44], atomicity violations [8, 26], order violations [10, 39, 50], and others. As we have discussed in details in earlier sections, DCatch draws inspiration from LCbug detection approaches and shares the same theoretical foundation (i.e., happens-before relationship) with LCbug detections. However, every component of DCatch design is customized to suit the unique need of DCbugs and distributed systems.

DCbugs and DCbug detection have not been well studied in the past. ECRacer [2] is an inspiring recent work that looks at applications using eventual consistency data stores. ECRacer generalized conflict serializability to the setting of eventual consistency and built a dynamic checker to examine whether a program execution is serializable considering the underlying eventual consistency data store. ECRacer and DCatch both look at concurrency problems in distributed systems. However, they target completely different problems. ECRacer focuses on how applications, such as mobile applications, use underlying distributed eventual-consistency data stores, while DCatch looks at general distributed systems, particularly infrastructure systems. ECRacer checks the serializability of the monitored run, while DCatch tries to predict DCbugs for future runs.

9. Conclusions

Distributed concurrency bugs (DCbugs) severely threat the reliability of distributed systems. They linger even in distributed transaction implementations [3, 24, 34]. In this paper, we designed and implemented an automated DCbug detection tool for large real-world distributed systems. The DCatch happens-before model nicely combines causality relationships previously studied in synchronous and asynchronous single-machine systems and causality relationships unique to distributed systems. The four components of DCatch tool are carefully designed to suit the unique features of DCbugs and distributed systems. The triggering module of DCatch can be used as a stand-alone testing framework. We believe DCatch is just a starting point in combating DCbugs. The understanding about false negatives and false positives of DCatch will provide guidance for future work in detecting DCbugs.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback and comments. This material is based upon work supported by the NSF (grant Nos. CNS-1563956, IIS-1546543, CNS-1514256, CCF-1514189, CCF-1439091) and generous supports from Huawei, Alfred P. Sloan Foundation, Google Faculty Research Award, and CERES Center for Unstoppable Computing.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. http://www.labri.fr/perso/casteran/CoqArt/index.html.

[2] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In *POPL*, 2017.

[3] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, 2007.

[7] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-Driven Programming. In *PLDI*, 2013.

[8] Cormac Flanagan and Stephen N Freund. Atomizer: A Dynamic Atomicity Checker For Multithreaded Programs. *ACM SIGPLAN Notices*, 39(1):256–267, 2004.

[9] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[10] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs. In *ASPLOS*, 2011.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP*, 2003.

[12] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI*, 2011.

[13] 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. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *SoCC*, 2014.

[14] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP*, 2011.

[15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *SOSP*, 2015.

[16] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[17] Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race Detection for Event-Driven Mobile Applications. In *PLDI*, 2014.

[18] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *ATC*, 2010.

[19] Baris Kasikci, Cristian Zamfir, and George Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, 2012.

[20] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[21] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[22] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

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

[24] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *ASPLOS*, 2016.

[25] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[26] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

[27] Brandon Lucia, Luis Ceze, and Karin Strauss. ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-Variable Atomicity Violations. In *ISCA*, 2010.

[28] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *SOSP*, 2015.

[29] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race Detection for Android Applications. In *PLDI*, 2014.

[30] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A Timely Dataflow System. In *SOSP*, 2013.

[31] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[32] Robert H. B. Netzer and Barton P. Miller. Improving The Accuracy of Data Race Detection. In *PPoPP*, 1991.

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

[34] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *ATC*, 2014.

[35] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Finding Places. In *ASPLOS*, 2009.

[36] Veselin Raychev, Martin T. Vechev, and Manu Sridharan. Effective Race Detection for Event-Driven Programs. In *OOPSLA*, 2013.

[37] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM TOCS*, 1997.

[38] Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, 2008.

[39] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. Do I Use the

Wrong Definition? DefUse: Definition-Use Invariants for Detecting Concurrency and Sequential Bugs. In *OOPSLA*, 2010.

[40] Benjamin H. Sigelman, Luiz Andr Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

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

[42] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic Recognition of Synchronization Operations for Improved Data Race Detection. In *ISSTA*, 2008.

[43] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.

[44] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and Surviving Data Races using Complementary Schedules. In *SOSP*, 2011.

[45] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Tom Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI*, 2015.

[46] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad Hoc Synchronization Considered Harmful. In *OSDI*, 2010.

[47] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.

[48] Jiaqi Zhang, Weiwei Xiong, Yang Liu, Soyeon Park, Yuanyuan Zhou, and Zhiqiang Ma. ATDetector: improving the accuracy of a commercial data race detector by identifying address transfer. In *MICRO*, 2011.

[49] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *ASPLOS*, 2011.

[50] Wei Zhang, Chong Sun, and Shan Lu. ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach. In *ASPLOS*, 2010.