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

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].

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

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
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] 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 ex-
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 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 $o_1 \Rightarrow o_2$. 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 $o_1 \Rightarrow o_2 \Rightarrow o_0 \Rightarrow o_2 \Rightarrow o_2 \ldots o_{k-1} \Rightarrow o_2$, $o_1$ 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.
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 DCG bug 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.

**RPCC** 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-MRPC**:

- **Create** \((r, n_1) \Rightarrow \text{Begin} \((r, n_2)\).
- **End** \((r, n_2) \Rightarrow \text{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-Msoc**:

- **Send** \((m, n_1) \Rightarrow \text{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.

### 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-Mpush**:

- **Update** \((s, n_1) \Rightarrow \text{Pushed} \((s, n_2)\).

Note that, this rule is not redundant given Rule-MRPC and Rule-Msoc. We can decompose this rule into three chains of causality relationship: (1) **Update** \((s, n_1) \Rightarrow \text{Recv} \((s, n_c)\); (2) **Recv** \((s, n_c) \Rightarrow \text{Send} \((s, n_c)\); (3) **Send** \((s, n_c) \Rightarrow \text{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-MRPC and Rule-Msoc 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 \( s \), 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-Mpull**:

- **Update** \((s, n_1) \Rightarrow \text{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

<table>
<thead>
<tr>
<th>App</th>
<th>Inter-Node</th>
<th>Intra-Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HBase</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MapReduce</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

![Figure 4. Concurrency and communication in MapReduce](image-url)
in Figure 4b. 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-Tfork:** \( Create(t) \xrightarrow{fork} Begin(t) \).

**Rule-Tjoin:** \( End(t) \xrightarrow{join} Join(t) \).

### 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-Eenq:** \( Create(e) \xrightarrow{enq} Begin(e) \).

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 Er:serial:** \( End(e_1) \xrightarrow{serial} Begin(e_2) \), 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 Pseq:** \( o_1 \xrightarrow{seq} o_2 \), 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 Pseq:** \( o_1 \xrightarrow{seq} o_2 \), 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: \( W \xrightarrow{reg} Create(t) \xrightarrow{fork} Begin(t) \xrightarrow{enq} Create(OpenRegion, HMaster) \xrightarrow{MPC} Begin(OpenRegion, HRS) \Rightarrow Create(e) \Rightarrow Begin(e) \Rightarrow Update(RS...OPENED, HRS) \xrightarrow{MPC} Pushed(RS...OPENED, HMaster) \xrightarrow{reg} R \).

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-MPC 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-Eenq.

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-
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 Javaassist 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 relationships 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) callback 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 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).
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 tens of 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., System.exit); (2) severe errors that are printed out, whose corresponding failure instructions are invocations of Log::fatal and Log::error functions in studied systems; (3) throwing uncatchable exceptions, such as 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 \texttt{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 our test cases execute \(A\) right before \(B\) and also \(B\) right before \(A\). We will simply put a \texttt{_request} API call before \(A\) and a \texttt{_confirm} API call right after \(A\), and the same for \(B\). At run time, the \texttt{_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 \texttt{_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 \texttt{_request} and \texttt{_confirm} APIs given a DCbug report \((s, t)\). The \texttt{_confirm} APIs can be simply inserted right after the heap access in the bug report. Therefore, our discussion below focuses on the placement of \texttt{_request} APIs.

The naive solution is to put \texttt{_request} right before \(s\) and \(t\). However, this naive approach may lead to hangs or too many \texttt{_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 \texttt{_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 \texttt{_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 \texttt{_request} in corresponding RPC callers. (3) If \(s\) and \(t\) are inside critical sections guarded by the same lock, DCatch suggests putting \texttt{_request} right before the corresponding critical sections.

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 \texttt{_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.

\textbf{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 \texttt{java.lang.Thread} interface. Event handling is implemented using \texttt{org.apache.hadoop.yarn.event.EventHandler} and \texttt{org.apache.hadoop.hbase.executor.EventHandler} interface in Hadoop and HBase. The prototype of an handler function is \texttt{EventHandler::handle} (\texttt{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, Proto-Record sages, EndPoint), 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 Jazzassist 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.

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

**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 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™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
have already been patched by developers in later versions. We have never discovered or patched and the remaining ware project, and found that two among these 8 DCbugs are no longer detectable. We have carefully checked the change log of each software project, and observed their harmful impact, such as node crashes and unavailable services, through DCatch triggering module. We were unaware of these bugs, and they are not part of the TaxDC bug suite. We have triggered all of them 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 truly harmful. The remaining 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.

### 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 t1 could delete a zknode concurrently with another thread t2 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 t2 would fail and cause HMaster to crash, if the delete from t1 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.

### 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 truly harmful. The remaining 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.

### 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 t1 could delete a zknode concurrently with another thread t2 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 t2 would fail and cause HMaster to crash, if the delete from t1 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.

### 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 truly harmful. The remaining 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.

### 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 t1 could delete a zknode concurrently with another thread t2 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 t2 would fail and cause HMaster to crash, if the delete from t1 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.

### 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 truly harmful. The remaining 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.

### 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 t1 could delete a zknode concurrently with another thread t2 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 t2 would fail and cause HMaster to crash, if the delete from t1 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.

### 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 truly harmful. The remaining 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.

### 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 t1 could delete a zknode concurrently with another thread t2 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 t2 would fail and cause HMaster to crash, if the delete from t1 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.
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 DCbugs. 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 unique 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.

### 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.
7.4 Comparison with alternative designs

Unselective memory-access logging  DCatch tracking 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 access-memory 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 effect. If DCatch ignored, then 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.

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<sup>RPC</sup>, Rule-M<sup>soc</sup>, and Rule-M<sup>push</sup>, 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<sup>reg</sup>. For example, when event-handler Begins and Ends 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 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<sub>1</sub> sends a message m<sub>1</sub> to node n<sub>2</sub>, and n<sub>2</sub> sends m<sub>2</sub> back in response. Tracing socket operations or not would both reach the conclusion that send m<sub>1</sub> happens before receive m<sub>2</sub> on n<sub>1</sub>, through different reasoning. Tracing socket operations or not would both reach the conclusion that send m<sub>1</sub> happens before receive m<sub>2</sub> on n<sub>1</sub>, through different reasoning. Tracing socket operations or not would both reach the conclusion that send m<sub>1</sub> happens before receive m<sub>2</sub> on n<sub>1</sub>, through different reasoning. Tracing socket operations or not would both reach the conclusion that send m<sub>1</sub> happens before receive m<sub>2</sub> on n<sub>1</sub>, through different reasoning.
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 nondeterministic 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., Pb [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 needs 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 Unstopabble Computing.

References


