FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems

Haopeng Liu
University of Chicago
haopliu@uchicago.edu

Xu Wang
Beihang University
wangxuact@gmail.com

Guangpu Li
University of Chicago
cstjygpl@uchicago.edu

Shan Lu
University of Chicago
shanlu@uchicago.edu

Feng Ye
Huawei US R&D Center
feng.fengye@huawei.com

Chen Tian
Huawei US R&D Center
chen.tian@huawei.comu

Abstract

It is crucial for distributed systems to achieve high availability. Unfortunately, this is challenging given the common component failures (i.e., faults). Developers often cannot anticipate all the timing conditions and system states under which a fault might occur, and introduce time-of-fault (TOF) bugs that only manifest when a node crashes or a message drops at a special moment. Although challenging, detecting TOF bugs is fundamental to developing highly available distributed systems. Unlike previous work that relies on fault-injection to expose TOF bugs, this paper carefully models TOF bugs as a new type of concurrency bugs, and develops FCatch to automatically predict TOF bugs by observing correct execution. Evaluation on representative cloud systems shows that FCatch is effective, accurately finding severe TOF bugs.

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

Keywords Timing Bugs; Fault Tolerance; Distributed Systems; Bug Detection; Cloud Computing

ACM Reference Format:
Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3173162.3177161

1 Introduction

1.1 Motivation

Distributed systems have become the backbone of computing, and their availability has become increasingly crucial — a cloud outage of a few minutes can easily cost a company millions of dollars [16, 42]. High availability of distributed systems largely hinges on how well common component failures [17], such as node crashes and message drops, can be tolerated and recovered from. Unfortunately, developers often cannot anticipate and hence would mis-handle some component failures, referred to as *fault*s, that occur at special moments and system states. We refer to the resulting type of bugs as time-of-fault bugs or TOF bugs.

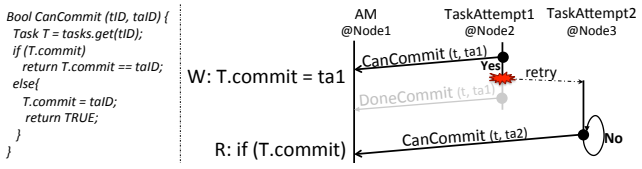

Figure 1. An example of TOF bugs from MapReduce

TOF bugs are among the most *severe* bugs in distributed systems. Many other bugs may cause a single node failure but not system failures due to fault-tolerance schemes built in distributed systems. Unfortunately, TOF bugs break fault-tolerance schemes and hence easily cause system failures.

TOF bugs are difficult to expose during in-house testing due to their complicated triggering conditions — component failures, which need to be injected during testing, have to occur under special timing. As a result, they widely exist in deployed distributed systems [9, 24, 37].

TOF bugs are also *unique* to distributed systems, as single-machine systems, except for storage systems, are not expected to tolerate node crashes.

Figure 1 illustrates a TOF bug from Hadoop-MapReduce. Here, a task attempt contacts Application Manager (AM) through an RPC `CanCommit` to get commit permission and get its attempt-ID `ta1` recorded. Soon later, it will inform AM that it has successfully committed through `DoneCommit`. Hadoop usually can tolerate a task-attempt crash: if the crash is after `DoneCommit`, no recovery is needed as the global state is consistent; if the crash is before `CanCommit`, another attempt `ta2` will redo the task. Unfortunately, if the attempt crashes between the two RPC calls, which is a small time window comparing with the attempt's whole life time, the job will never finish. The reason is that `T.commit` on AM has been contaminated by the crashed attempt `ta1`, causing every recovery attempt to fail the `CanCommit` checking.

As we can see, the complexity of TOF bugs is inherent to distributed systems, where nodes interact with each other through global files and messages. When a node N_{Crash} crashes, its remaining states scatter around the system, including global files updated by N_{Crash}, remaining nodes' heaps updated by RPC functions remotely invoked by N_{Crash} (e.g., `T.commit` on AM in Figure 1), and others. Depending on which node crashes at which point of the system execution, different system states could be left behind and demand different handling from the remaining live nodes and the

ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3177161

recovery routine. Searching the huge state space of a distributed system for small time windows where a particular node's crash is improperly handled is a daunting task.

The state of practice in catching TOF bugs is ineffective. It injects faults randomly, hoping to hit small bug-triggering time windows through many tries. Previous work [11] and our own experiments all show that real-world TOF bugs often do not manifest even after hundreds of such random fault-injection runs with bug-triggering workload.

Recent research improves the efficiency of fault-injection testing, leveraging manual specification or expert knowledge of the system under test. Distributed-system model checkers use heuristics, like injecting faults only at message sending/receiving points, and rules specified by developers to avoid some unnecessary fault injections [23, 30, 36, 51, 56]. Lineage-driven fault injectors [10, 11] require system models manually written in a domain specific language. Some recent effort relies on domain experts to decide which part of the system is important and hence should take fault injection [9, 19, 20].

Although recent work has achieved great progresses, they all require much manual effort and/or deep understanding of the system under test. Furthermore, they still struggle at searching through the huge state space of distributed systems, with most (e.g. >99%) of their carefully designed fault injections not revealing any TOF bugs [9, 19, 20, 36].

1.2 Our Contributions

Different from previous work that relies on random or manually-guided fault injections to *hit* TOF bugs, our tool FCatch uses program analysis to automatically *predict* TOF bugs with high accuracy, without manual specification or domain knowledge about the software under test. That is, by observing a correct execution with no faults or a successfully recovered fault at time t, FCatch can predict that the system execution would fail when a fault occurs at t' ($t' \neq t$).

A new TOF bug model FCatch is built upon a new model of TOF bugs. Different from previous work that essentially models TOF bugs as semantic bugs, and hence requires semantic specifications in bug detection. We model TOF bugs as a type of concurrency bugs, with two key properties:

- Triggering conditions: a TOF bug is triggered by a special timing of fault f. If the fault f, like the task-attempt crash in Figure 1, occurs a bit earlier or later, the system would behave correctly.
- Root causes: once triggered, a TOF bug can cause system failures when the faulty node N_{Crash} leaves a shared resource in a state that cannot be handled by another node N. For example, the failure in Figure 1 is caused by the original task attempt leaving a non-NULL `T.commit` in AM that cannot be handled by the recovery attempt.

Starting from these two key properties, other detailed properties of TOF bugs can then be reasoned about and guide TOF bug detection, which we will describe in Section 2.

A new TOF bug detection approach This model provides new opportunities to detect TOF bugs.

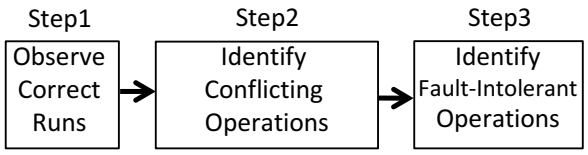

Figure 2. The flow of FCatch

- The triggering conditions suggest that we can predict TOF bugs by analyzing correct runs to see whether/how the system might behave differently when the time of fault changes. This approach uses runs under common timing and workload as correctness specifications, greatly reducing manual effort in TOF bug detection.
- The root causes suggest that we can predict TOF bugs by analyzing operations that read and write the same resource from different nodes, referred to as *conflicting* operations. This approach can help greatly shrink the TOF bug search space, avoiding unnecessary checking.

FCatch Following the above observations and approach, we build FCatch that predicts TOF bugs in three main steps, as shown in Figure 2.

First, FCatch monitors correct fault-free or faulty runs of a distributed system, tracing resource-access operations and fault-tolerance related operations. Particularly, FCatch carefully designs which correct runs to monitor — a TOF bug like that in Figure 1 requires monitoring and comparing more than one correct run to discover — and what to trace for each run. The details are presented in Section 3.

Second, FCatch analyzes traces to identify pairs of conflicting operations that write and read the same resource, such as data in heap or persistent storage, from different nodes. Particularly, following our TOF bug model in Section 2, FCatch adapts traditional happens-before analysis to identify every pair of conflicting operations whose interaction can potentially be perturbed by the time of fault. The details are in Section 4.

Third, FCatch analyzes the distributed system to identify conflicting operations that are not protected by existing fault-tolerance mechanisms such as timeouts, sanity checks, and data resets. These operations are referred to as fault-intolerant operations. They are reported by FCatch as TOF bugs. The details will be presented in Section 4.

Finally, FCatch tries to trigger every TOF bug reported above, helping developers confirm which reported bugs can truly cause failures. The details are presented in Section 5.

We evaluated FCatch using a set of 7 TOF bugs collected by an existing benchmark suite of real-world distributed-system concurrency bugs [37]. These 7 TOF bugs come from 4 widely used distributed systems, Cassandra, HBase, MapReduce, and ZooKeeper, and can be triggered by 6 common workloads under special time of faults.

By analyzing only one or two **correct** runs of each workload, FCatch generates 31 TOF bug reports. These include 8 reports that explain the 7 benchmarks, 8 reports[1] that are truly severe TOF bugs

[1]Detailed readmes and reproduction scripts of all the true TOF bugs can be accessed at https://github.com/haopeng-liu/TOF-bugs.

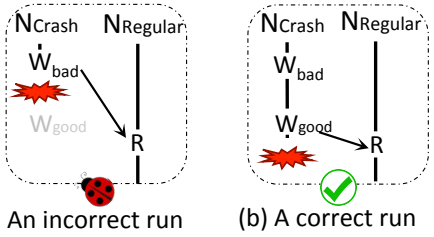

Figure 3. Conflicting operations in a crash-regular bug. →: define-use relation; gray items disappear due to crash.

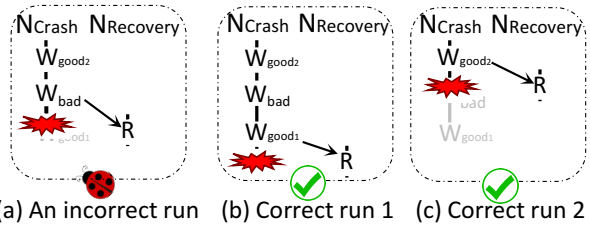

Figure 4. Conflicting operations in a crash-recovery bug. →: define-use relation; gray items disappear due to crash.

beyond the initial benchmark suite [37], which we were unaware of before our experiments, 6 that cause well-handled exceptions, and 9 that are benign. All the false positives can be easily pruned by FCatch's automated bug-triggering module. In comparison, only one out of all these bugs could be exposed after 400 fault-injection runs of corresponding bug-triggering workload, and even this one bug has less than 3% of manifestation rate. FCatch bug detection introduces 5X – 15X slowdown to the baseline fault-free execution, suitable for in-house testing.

2 Modeling TOF bugs

We categorize TOF bugs into two types: (1) crash-regular bugs, where conflicting operations are from the crash node N_{Crash} and a non-crash node N_{Regular}, illustrated in Figure 3; (2) crash-recovery bugs, where conflicting operations are from the crash node N_{Crash} and the recovery node N_{Recovery}, exemplified in Figure 1 and illustrated in Figure 4.

In the following, we present our detailed models of these two types of TOF bugs, which guide the design of FCatch.

Terminology We refer to *faults* as component, rather than system, failures that need to be tolerated. Among different types of faults, we focus on node crashes and message drops. We use *faults* and *crashes* interchangeably, unless otherwise specified. When deploying a distributed system, one can configure a process to run by itself or co-located with other processes in a physical node. Therefore, we use *node* and *process* interchangeably. When we say an operation O is *from* a node N, O could be physically executing on N or causally initiated by N, like in an RPC function remotely invoked by N. Section 4.1 presents how we analyze such causal relationships. The *happens-before* relationship discussed below is the same logical timing relationship discussed in previous work [33, 39], which we will elaborate in Section 4.1.

2.1 Crash-regular TOF bugs

What are these bugs? As shown in Figure 3a, by definition, a crash-regular bug manifests when a crash causes a regular node N_{Regular} to read, denoted by R, a shared resource defined by an unexpected source W_{bad} in node N_{Crash}.

Thinking about why the time of fault could make W_{bad} an unexpected source for R reveals more information:

1. The expected source for R, denoted as W_{good}, must execute after W_{bad} from N_{Crash} during correct runs (i.e., Figure 3b), and consequently could disappear due to untimely crashes in incorrect runs (i.e., Figure 3a). Otherwise, if W_{good} executes before W_{bad}, it would not be affected by any crashes after W_{bad} and hence leaves no chances for any crashes to make R read from W_{bad} instead of W_{good}.
2. W_{good} must always execute before R during correct runs. That is, W_{good} must *happen before* R. Otherwise, if it is concurrent with R, W_{bad} could become a source for R even without untimely crashes.
3. The disappearance of W_{good} must block the execution of R (e.g., the missing of a `signal` causing a `wait` to block forever), leading to failures. As we will discuss in Section 4.1, given that W_{good} happens before R, the only other possibility is that W_{good}'s disappearance causes the whole thread/function of R to disappear (e.g., the missing of an RPC call causing the RPC-handler not to execute), which is a natural consequence of the node crash yet not a bug.
4. There must be no fault-tolerance mechanism that can unblock R. For example, with a timeout, a common fault-tolerance mechanism, the above bad interaction between W_{bad} and R would have been tolerated.

This gives us a profile of a crash-regular TOF bug. It is related to a pair of conflicting operations $\{W, R\}$. R consumes the state of a shared resource, which is a heap object on a non-crash node or persistent storage data anywhere, defined by W during correct runs (i.e., W_{good} in Figure 3). When the node, which W comes from, crashes before W, R's execution is blocked forever, causing hangs and related failures.

How to detect these bugs? With the above model, instantiating the general flow in Figure 2 to detect crash-regular TOF bugs is straightforward.

First, FCatch monitors a fault-free run of the target system, recording resource-access operations, happens-before operations, and time-out operations (Section 3).

Second, FCatch identifies pairs of conflicting operations from different nodes that have blocking happens-before relationship with each other (Section 4.2.1).

Third, FCatch checks time-out mechanisms in the target system and reports conflicting and fault-intolerant crash-regular operation pairs as candidate bugs (Section 4.2.2).

Finally, FCatch tries to trigger a reported {W, R} crash-regular TOF bug by crashing the node of W right before W or dropping the message where W is from (Section 5).

2.2 Crash-recovery TOF bugs

What are these bugs? By definition, crash-recovery bugs differ from crash-regular bugs in where the read operation R comes from. In other words, they differ on whether the impact of the crash is

incorrectly handled by the recovery node N_{Recovery} or a regular node N_{Regular}.

This seemingly small difference fundamentally changes the define-use relationship for R. When R is in a regular node, what resource content it consumes is affected by the happens-before relationship and program synchronization. For example, given an update W in node N_{Crash} and that R happens before W, W can never define the content consumed by R no matter when the crash occurs. However, when R is in a recovery node, what resource content it consumes is *completely* determined by the time of crash. Traditional happens-before relationship and program synchronization can*not* affect when N_{Crash} crashes and hence can*not* regulate when N_{Recovery} starts.

With this in mind, we think about why the time of fault could make W_{bad} an unexpected source for R:

1. The expected source for R could execute either after W_{bad} (Figure 4b) or before W_{bad} (Figure 4c) during a correct run, when the fault occurs later or earlier than that during the incorrect run.
2. There must be no fault-tolerance mechanism that can prevent R from consuming content defined by W_{bad}. Analyzing and identifying this type of fault-tolerance mechanisms is much more complicated than identifying time outs, and involves both data-dependence and control-dependence analysis, which we will discuss in details in Section 4.3.2.

This gives us a profile of a crash-recovery TOF bug. It is related to a pair of conflicting operations {W, R}, with W from N_{Crash} and R from N_{Recovery}. N_{Recovery} tries to recover the crash of N_{Crash}. Depending on the time of the crash, the recovery attempt sometimes succeeds, with R consuming shared-resource content defined by W, and sometimes fails, with R consuming shared-resource content defined after W or before W.

How to detect these bugs? We follow the above model to instantiate a crash-recovery bug detection flow.

The first step is more challenging than that for crash-regular TOF bugs because we may need to observe both a fault-free run and a faulty run, and stitch them together. If only observe a fault-free run, we cannot possibly know what are the recovery operations like R. If only observe a correct faulty run like the one in Figure 4c, we cannot know what operations, including W_{bad}, could have happened if the node crashed later. Essentially, we may need to predict a TOF bug {W, R}, with its W observed in one run and its R observed in another. Section 3 will discuss our solutions.

At the second step, we identify every pair of conflicting operations {W, R}, so that W from node N_{Crash} defines the shared-resource content consumed by R from node N_{Recovery} during a correct faulty run, together with an alternate W' that updates the same shared resource right after or right before W from N_{Crash} (Section 4.3.1).

Third, we conduct control-flow and data-flow analysis to prune out a bug candidate {W, R}, if fault-tolerance mechanisms in software prevent R from consuming content defined by the alternate update W' (Section 4.3.2).

Finally, FCatch tries to trigger a reported crash-recovery bug {W, R} by crashing the node of W right before or after W (Section 5).

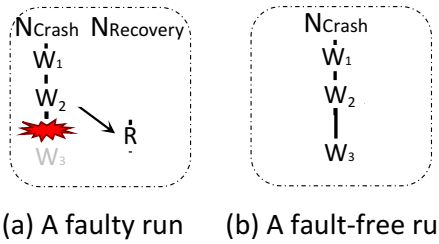

Figure 5. Observe two runs to detect a crash-recovery bug

2.3 Discussion

Our model above does not cover all bugs that are related to faults in distributed systems. We only cover node-crash and some message-drop faults. We do not cover bugs that are related to faults, but not the timing of faults. For example, some exception handlers are incorrectly implemented and would lead to failures every time they are invoked. We also do not cover bugs that may involve the interaction among more than one fault or more than one resource. Previous work on detecting multi-variable TOE bugs could help extend FCatch to tackle these bugs.

3 Tracing Correct Runs

This section discusses how FCatch monitors correct system execution and generates traces. The next section will explain how FCatch analyzes these traces to detect TOF bugs.

3.1 Which correct runs to observe?

As mentioned in Section 2.1, to predict crash-regular TOF bugs, FCatch can trace any correct run. However, to predict crash-recovery TOF bugs like the one in Figure 1, the tracing is much more challenging and hence is the focus below.

The first challenge is that conflicting operations of a crash-recovery bug may *never* appear together in the same correct run. Consequently, no dynamic detection tool can possibly predict such a bug by observing just one correct run, a unique challenge that does not exist in traditional concurrency-bug detection.

For example, in Figure 1, after AM executes the first `CanCommit` RPC for a task T, it will never return `TRUE` for `CanCommit` invoked by other attempts on T. Since both conflicting operations are in `CanCommit`, once one executes, the other can only execute in a task attempt that cannot finish due to the repeated `FALSE` returns from `CanComit`. At that time, it is already too late to *predict* the bug.

To address this challenge, we predict a crash-recovery bug based on not one correct run, but two complementing correct runs, a faulty one and a fault-free one under the same workload as shown in Figure 5. From these two runs, we can observe not only what happened on N_{Crash} prior to the crash (e.g., W_1 and W_2 in Figure 5a) and what happened in the recovery routine on N_{recovery} based on the faulty run (e.g., R in Figure 5a), but also what *could have* happened on N_{crash} if the fault occurred later based on the fault-free run (e.g., W_3 in Figure 5b).

Take the bug in Figure 1 as an example. To predict this bug, we observe two runs: one has the attempt crashing *anywhere* before `CanCommit`, which reveals the `T.commit` checking from the recovery attempt, and one has the attempt finishing successfully, which reveals the setting of `T.commit` from the initial attempt.

However, **a second challenge** comes: how to stitch traces from two runs together. Since an object's hash code changes from run to run, we cannot tell whether an operation from one run, like the write on `T.commit` in a fault-free run, accesses the same object as an operation from another run, like the read of `T.commit` in a faulty run. Furthermore, given the inherent non-determinism in distributed systems, the two runs likely already diverged prior to the crash point. Consequently, naively using the fault-free run to predict what might happen if the fault occurs later in the faulty run might introduce a lot of inaccuracies in TOF bug detection.

Our solution leverages virtual machines' checkpointing mechanism, a standard technique supported by all types of virtual machine (VirtualBox, VMWare, etc.) and container (Docker) systems. The high-level idea is to run the whole distributed system inside a Virtual Machine, and take a whole system checkpoint at a (random) point. We then make the system resume from the checkpoint in two ways, one without any fault injection and the other one injects a node crash immediately after the checkpoint. After both runs finish the workload, we will get a pair of perfectly complementing runs. These two runs share exactly the same execution prefix and heap object layout prior to the checkpoint. Consequently, we completely eliminate the execution non-determinism problem prior to the crash, and we can simply use object hash code to identify conflicting operations across the two runs. This idea can be implemented on any virtual machine infrastructure. Our current implementation is built upon VirtualBox Virtual Machine Monitor [45] and we will explain the details in Section 6.

Note that, although the scheme above requires a faulty run, it is fundamentally different from previous fault-injection testing. FCatch needs a correct faulty run, which is abundant in a mature distributed system, to predict TOF bugs; previous fault-injection testing needs an incorrect faulty run, which is naturally rare, to observe a TOF bug. To detect the bug in Figure 1, FCatch needs a run where the attempt crashes before `CanCommit`. Since an attempt spends most of its time before `CanCommit`, almost every random fault injection works.

3.2 What to trace?

FCatch generates one trace for every thread in system. It traces two sets of operations. The first set includes operations that affect intra-node and inter-node causal and blocking relationship, such as thread creation/join, signal/wait, event operations, message, RPC, etc (more details in Section 4.1). The second set includes all candidates for *conflicting operations*, which are operations that access two types of shared resources.

The first type of resources are heap objects on a non-crashing node, like `T.commit` on AM in Figure 1. Specifically, FCatch traces all heap accesses inside message handlers, RPC functions, event handlers and their callee functions, which are most related to inter-node communication and computation. FCatch does not trace all heap accesses in a system so that its dynamic tracing and trace analysis can scale, which we will evaluate in Section 8.2. Note that, FCatch will not consider heap objects on the crash node in its analysis, as their content will be wiped off by the crash and hence have no impact to the remaining system.

The second type of resource is persistent data in file systems, key-value stores, etc. Specifically, FCatch traces operations on local and global files (e.g., HDFS files), and records in distributed key-value stores (ZooKeeper ZKnodes). We consider create, delete, read, write, rename, and check-if-exist APIs for each storage system.

Every record of a traced operation consists of four parts: (1) operation type, specifying which type of operations on which type of resources; (2) callstack; (3) local time stamp counter obtained by `RDTSCP`, which provides an approximate nano-second level ordering among operations in one machine; (4) ID, which uniquely identifies every shared resource and every message/RPC/event. We use JVM hash code for heap object ID[2]; use path string for file ID; use random integer inserted as extra parameter (field) for RPC (message) ID.

4 Predicting TOF bugs

4.1 Causal and Blocking Relationship Analysis

We first discuss two sub-types of happens-before relationship, referred to as causal and blocking relationships. The latter helps FCatch detect crash-regular TOF bugs; the former helps FCatch judge whether an operation that physically executes on a node N logically comes from another node N', which is used in detecting both crash-regular and crash-recovery TOF bugs.

Causal relationship and blocking relationship We define an operation A to *causally depend* on an operation B, denoted as $B \xrightarrow{c} A$, if the disappearance of B causes A to also disappear. For example, the disappearance of a message-sending operation causes operations inside the message handler to also disappear. Given $B \xrightarrow{c} A$, we consider A to logically come from or get initiated by the node N_B that B physically executes on.

We define an operation A to *blocking depend* on B, denoted as $B \xrightarrow{b} A$, if the disappearance of B causes the thread that contains A to hang. For example, the disappearance of a signal operation causes the thread conducting the wait operation to hang.

Causal operations and blocking operations We take distributed systems' common happens-before operations [39], and separate them into causal and blocking operations.

FCatch considers all common intra-thread (event-based), inter-thread, and inter-node causal operations in distributed systems: (1) event enqueue. $EnQ(e) \xrightarrow{c}_d o$ where o is any operation inside the handler of `e` and its callee functions ($\xrightarrow{c}_d$ denotes *direct* causal relationship); (2) thread creation. $create(t) \xrightarrow{c}_d o$, where o is any operation inside thread `t`; (3) RPC invocation. $call(R) \xrightarrow{c}_d o$, where o is inside RPC function `R` and callees; (4) message sending. $send(m) \xrightarrow{c}_d o$, where o is inside the message handler of `m` and callees; (5) a state update through a synchronization service like ZooKeeper. $update(s) \xrightarrow{c}_d notify(s)$, where $notify(s)$ is the corresponding notification operation.

Given $B \xrightarrow{c}_d A$, we call A as the *Causee* of B and B the *Causor* of A. An operation has at most one causor. A causal operation may have more than one causee.

FCatch considers two common types of blocking operations: standard condition-variable signal and custom while-loop signal. The disappearance of the former would block corresponding waits and the waiting threads. The disappearance of the latter would block a corresponding while loop.

[2]Object::hashCode() is a practical method provided by JVM [8] to return distinct integers for distinct objects.

Causality analysis Causal relationship is transitive. If $B \xrightarrow{c} A$ and $C \xrightarrow{c} B$, then $C \xrightarrow{c} A$. Since FCatch traces the execution of all causal operations, it is straightforward to identify all the operations $\mathbb{O}_{\mathbb{S}\rightarrow}$ that causally depend on a set of seed operations $\mathbb{S}$ (Algorithm 1), and to identify all the operations $\mathbb{O}_{\rightarrow\mathbb{S}}$ that a set of seed operations $\mathbb{S}$ causally depend on (Algorithm 2). The former will be used to identify operations that logically come from N_{Crash} and N_{Recovery}, respectively (Section 4.3.1), and the latter will be used to identify operations that physically execute on one node yet logically come from another (Section 4.2.1).

```
Input: A set of seed operations 𝕊
Output: A set of operations 𝕆_{𝕊→} that causally depend on 𝕊
Stack<Op> WorkingSet = 𝕊;
Set<Op> VisitedSet = Null;
Set<Op> 𝕆_{𝕊→} = Null;
while h = WorkingSet.pop() do
    if h.type == Causal_Operation then
        while c = Causee(h).pop() do
            if VisitedSet.Add (c) then
                WorkingSet.push (c);
        end
    𝕆.push (h);
end
return 𝕆;
```

Algorithm 1: Identify operations causally depending on $\mathbb{S}$

```
Input: A set of seed operations 𝕊
Output: A set of operations 𝕆_{→𝕊} that causally cause 𝕊
Stack<Op> WorkingSet = 𝕊;
Set<Op> VisitedSet = Null;
Set<Op> 𝕆_{→𝕊} = Null;
while h = WorkingSet.pop() do
    if VisitedSet.Add (Causor(h)) then
        WorkingSet.push (Causor(h));
    𝕆.push (h);
end
return 𝕆;
```

Algorithm 2: Identify operations $\mathbb{S}$ causally depending on

4.2 Predicting Crash-Regular TOF bugs

A crash-regular TOF bug is triggered by unexpected interaction between a crashing node and a regular node. Figure 6 illustrates a bug from HBase, a widely used key-value store system. When a RegionServer (RS) starts to open the `Meta` table, it registers `OPENING` with ZooKeeper, which then causes a `Meta` record to be added in the `RIT` map. The master node HMaster then waits for RS to finish by repeatedly checking `RIT` in a loop, denoted as R in Figure 6. When RS finishes, it registers `OPENED` with ZooKeeper, which through a chain of events causes the `Meta` record to disappear from `RIT`, denoted as W in figure. W helps HMaster to jump out of its loop. Unfortunately, if RS crashes after registering `OPENING` and before registering `OPENED` with ZooKeeper (or if the `ToOpened` message drops), HMaster will hang in the while-loop and make the **whole** HBase system unavailable

This type of TOF bugs also have small triggering time-windows, and hence are difficult to expose during testing. In Figure 6, the failure-triggering fault window, denoted by an orange stripe, only lasts while RS creates two HDFS files and a ZooKeeper znode, a small window comparing with the whole HBase start-up and request-serving time. If RS crashes outside this window, HMaster will correctly re-assign the `Meta` table to another region server without hangs.

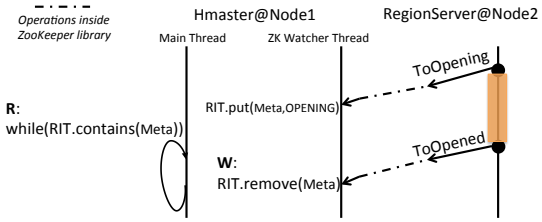

Figure 6. An example of TOF bugs from HBase

As discussed in Section 2, FCatch predicts this type of bugs by analyzing a correct-run trace in two steps.

4.2.1 Identifying conflicting operations

Is one blocking the other? FCatch first identifies W–R pairs that satisfy blocking relationship $W \xrightarrow{b} R$, as defined in Section 4.1. That is, FCatch needs to identify standard and custom signal-wait pairs, where the disappearance of the signal would block the thread/handler of the wait.

Identifying the former is straightforward. FCatch records every signal/wait right before it is invoked with a timestamp and an ID of the corresponding condition variable (CV). When processing the trace, FCatch orders all the `signal` and `wait` operations on a CV based on their timestamps and matches each `wait` with the first `signal` after it.

Identifying the latter follows state-of-art techniques in finding custom synchronization loops [52, 55]. We first use static analysis to identify likely-synchronization loops and heap reads that affect the exits of these loops through control and data dependency analysis (details in Section 6). FCatch traces all these reads dynamically. During trace analysis, FCatch identifies every heap write W whose update is used by at least one such heap read R from a different thread or handler, based on time-stamp and object hash-code comparison. This pair of W and R is then treated as a pair of custom signal and wait.

Are they from different nodes? Every pair of TOF bug candidate identified above, W and R, physically executes on the same node N. Next, we check whether W might logically come from another node, whose fault can cause R to hang.

We simply use Algorithm 2 to find all operations that W causally depends on. Once we find an operation W', $W' \xrightarrow{c} W$, that physically executes on a different node N', we consider W and R to pass the checking and will report them as a TOF bug— once W' disappears due to a node crash or a message drop, W would disappear and the thread holding R on N would hang. For example, while analyzing W in Figure 6, back tracking along the arrows eventually brings us to the RegionServer. Consequently, we know that a fault on the RegionServer will cause HMaster to hang. This analysis cannot be done statically, as the dynamic context could decide whether a function and hence operations within it are invoked by a local thread or a remote RPC call.

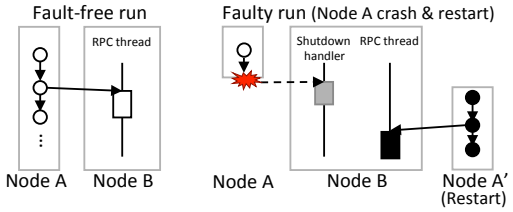

Figure 7. Crashing (white) and recovery operations (black, gray).

```
//crash operations
f.write(); //W: take snapshot
...
```

```
//recovery operations during reboot
DataTree dt = null;
for (File f : dir.sortedSnapshots())
 if (f.valid()) { //R1
  dt = f.read(); //restore; R2
  break;
 }
```

Figure 8. An example of sanity check in Zookeeper

4.2.2 Checking fault-tolerance mechanisms

FCatch statically checks whether there are time-outs that prevent R from waiting for the corresponding W forever. Given a signal-wait TOF bug candidate, we simply check whether the wait API is associated with timeout parameters, such as `Object::wait(long timeout)` instead of `Object::wait(void)`. Given a while-loop TOF bug candidate, we first check whether there exists a system-time acquisition function such as `System.currentTimeMillis` inside the loop body, and then check whether the time variable affects the loop exit condition through control or data dependence.

Of course, our current implementation may miss some time-out mechanisms, such as when dedicated monitoring threads are used to terminate a hanging loop or a wait.

4.3 Predicting Crash-Recovery TOF bugs

Crash-Recovery TOF bugs are caused by interaction between the crashing node and the recovery node, like in Figure 1. Given a pair of traces from a faulty run, where N_{crash} crashes at time T_C, and a complementing fault-free run, FCatch will report conflicting operations that might cause the system to misbehave if the crash occurs before or after T_C on N_{crash}.

4.3.1 Identifying conflicting operations

FCatch needs to identify operation pairs $\{W, R\}$ that access the same shared resource from the crashing node and the recovery node.

To identify operations that access persistent resources *physically* on the crashing or recovery node is trivial.

To identify operations that causally come from N_{Crash} (N_{Recovery}) but access persistent or heap resources outside it, we simply use all the RPC invocations and message sendings that escape N_{Crash} (N_{Recovery}) as seed operations and feed them into Algorithm 1.

Once all the recovery operations and crash operations are identified from the faulty trace and the fault-free trace respectively, finding pairs of them that access the same resource simply requires comparing corresponding resource IDs in the traces. Benefiting from how we obtain the perfectly pairing traces, no resource-ID translation is needed across traces[3].

So far, we have not discussed what exactly is the *recovery node*. In practice, it could be a restarted node that tries to recover from its recent crash, which happens in almost all systems; it could be a new process launched on a remaining node, which happens in MapReduce; it could also be a node-crash handler running inside an existing process through a unified interface, such as `IFailureDetection EventListener::convict` in Cassandra and `ZooKeeperListener ::nodeDeleted` in HBase, or a diverse set of functions in MapReduce and ZooKeeper.

In theory, FCatch can work with all these cases. By default, FCatch only treats processes that exist in the faulty trace but not in the fault-free trace as recovery nodes, obtained by process ID comparison (e.g., node A' in Figure 7). All operations in these processes or causally depend on these processes (e.g., all black shapes on node A' and B in Figure 7) are considered recovery operations. If developers specify recovery-handler interfaces or functions, FCatch can identify more recovery operations, such as the gray operations in Figure 7.

4.3.2 Checking fault-intolerance mechanisms

We generalize fault-tolerance mechanisms that can help prevent crash-recovery TOF bugs into two types: (1) the ones that use sanity checks to allow R skipping the consumption of unexpected content, and (2) the ones that update the content of a shared resource at the beginning of a recovery routine to make sure that no left-over content can reach R. We identify these two types of fault-tolerance mechanisms through the following control-dependency and data-dependency analysis.

Control-dependency analysis The recovery node may conduct sanity checks to decide which shared resources it will access and how, which helps prevent many crash-recovery TOF bugs. Figure 8 shows a simplified example from ZooKeeper. Here, a restarted ZooKeeper node checks the length and checksum segment of snapshot files, denoted by `f.valid()`, so that it can use the latest consistent snapshot file to recover its `DataTree`. This sanity check, denoted by `R1` in figure, prevents the ZooKeeper recovery routine from consuming invalid snapshot files left by the crash node.

To identify the above fault-tolerance mechanism, we first generalize it into the following pattern. There are two pairs of conflicting crash-recovery operations (i.e., W–R_1 and W–R_2 in Figure 8). The two recovery operations (i.e., R_1 and R_2 in figure) have control dependence with each other. Consequently, one serves as the fault-tolerance sanity check for the other.

Following this pattern, FCatch goes through every pair of recovery operations R_1 and R_2 touching the same resource, with each involved in any least one crash-recovery TOF bug candidate. If FCatch finds R_2 to control depend on R_1, all conflicting-operation pairs that contains R_2 are pruned.

All the dependency checking in FCatch is implemented in WALA. We check both intra-procedural and inter-procedural dependence following the call-stack recorded in trace where R_1 affects the return value of a function in its call-stack and this return value affects whether R_2 executes.

Data-dependency analysis The recovery node sometimes resets the content of a shared resource before using it. This way, later reads

[3]There could be inaccuracy if the crash op. and the recovery op. both access a heap object created after the crash point, but this is very rare.

will not consume content left by the crashing node. For example, in MapReduce, every submitted job is associated with a state variable and a job-report URL string. When an AM crashes, the recovery handler in RM immediately resets the URL string to `null` and the state to `INIT`.

To identify this type of fault-tolerance mechanisms, given a recovery operation R that is part of a crash-recovery TOF bug candidate, we search through all traced recovery operations for a write operation that accesses the same resource as R and executes before R judged by the causality relationship or their timestamps. If such a write operation exists, we prune out any TOF bug candidate that contains R.

4.3.3 Impact estimation

Not all conflicting fault-intolerance pairs identified above can cause harms. FCatch statically estimates the impact of each pair and prunes out the ones that are likely benign.

Specifically, given an operation R observed in a correct run, FCatch statically checks whether R might lead to any signs of potential harms either locally or globally because of consuming unexpected content. FCatch considers R to have a failure-prone local impact if it affects an exception `throw`, the printing of a severe error (i.e., fatal-level log printing), the creation of an event, and the startup of a service through data or control-dependence. FCatch considers R to have a failure-prone global impact if it can affect the return value of an RPC function or an RPC/message invocation/sending through data or control dependency. If FCatch finds that R has no failure-prone impact either locally or globally, FCatch prunes all the races that contain R out from the final TOF bug reports, similar as previous work on distributed concurrency bug detection [39].

5 TOF bug triggering

FCatch tries to automatically trigger every FCatch bug report, so that we can observe whether the reported bug is harmful or not.

Each report of a crash-regular TOF bug contains a triplet (W, R, W'). W and R physically execute on one node; W' is an RPC/message sending operation physically located on another node; $W \xrightarrow{b} R$ and $W' \xrightarrow{c} W$. To trigger this bug, FCatch tries injecting three types of faults right before W': 1) node crash emulated by `Runtime.get Runtime.halt(-1)`; 2) kernel-level message drop due to network-connection broken, emulated by skipping W' and throwing `java.net.SocketException`; 3) application-level message drop emulated by skipping W'. This only applies for Cassandra's droppable messages, which are dropped if staying for too long in the sending queue. We inject a special tag for the W' message, so that the networking-related code can differentiate this message from others and apply fault-injection correctly. We will evaluate the difference between these different types of faults in Section 8.4.

Each report of a crash-recovery TOF bug contains a pair (W, R), with W from N_{crash} and R from N_{recovery}. If W was already observed by FCatch during a correct faulty run, FCatch triggering script injects a node crash right before W to check what might happen if W does not appear due to the crash. If W was not already observed in a faulty run but only observed by FCatch in a fault-free run, FCatch triggering script injects a node crash right after W to check what might happen if W appears prior to the node crash.

```
1 void snapshotANDcrash(Runtime r){
2   r.exec("ssh_host_snapshot.sh");
3   sleep(5);
4   bool crash_flag = r.exec("ssh_host_read_flagfile");
5   if (crash_flag == "crash") {
6     r.halt(-1);
7   }
8 }
```

Figure 9. Our snapshot-and-crash wrapper

6 Implementation Details

Code Instrumentation FCatch implements its code instrumentation in Javassist [28], a Java byte-code transformation tool, and its static analysis in WALA [27], a static Java analysis framework. Specifically, FCatch uses WALA to statically identify functions to instrument following super-class interfaces like `VersionedProtocol` interface for RPC, `java.Thread.thread` for thread functions, `org.apache.hadoop.yarn.event.EventHandler` for event handling, etc. FCatch uses Javassist to automatically insert tracing functions before heap accesses (i.e., `getfield/putfield` instructions) and static-variable accesses (i.e., `getstatic/putstatic` instructions).

Virtual machine checkpointing Once randomly decide a crash point in the program, we insert a snapshot-and-crash wrapper function into the program, as shown in Figure 9. We then run the distributed system inside a VirtualBox VM. Before the crash point is reached, our `snapshotANDcrash` function invokes a script in the host OS to execute a VM-snapshot command asynchronously. The execution then reaches the `sleep` statement, which helps make sure that the snapshot is taken in the middle of the sleep. After the sleep, the original execution reaches Line 5 in Figure 9. There, a configuration file will decide not to inject a node-crash fault, and the execution will eventually finish as the *fault-free run*. After the *fault-free run*, we change the `flagfile` to allow fault injection, and then launch the Virtual Machine again starting from the snapshot recently taken. This time, the program resumes from inside the `sleep`, reaches the reading of `flagfile`, and eventually triggers a node crash through `r.halt(-1)` (Line 6 in Figure 9). The system continues after the node crash and contributes a *fault run* for FCatch bug detection. We currently emulate a node crash by terminating a process through `halt(-1)`. If we want to emulate crashing a multi-process node, we can simply `halt` multiple processes accordingly.

Synchronization loop identification To detect crash-regular TOF bugs, the tracing component of FCatch statically identifies loops satisfying the following two conditions as likely synchronization loops. First, the loop is not bounded on a constant value or the size of a container (e.g., `for(element e: list)` and `while(iterator.hasNext())`). Second, the loop body relinquishes the CPU through APIs like `Thread::sleep` and `Object::wait`.

FCatch false positives and negatives Like most bug detectors, FCatch is neither sound nor complete for several reasons. (1) Bug modeling. FCatch does not cover bugs beyond our TOF bug model (Section 2.3). (2) (no) Specifications. FCatch eases TOF bug detection by not relying on any manual specifications. Inevitably, this could cause inaccuracies. FCatch may miss recovery operations as discussed in Section 4.3.1, and may ignore true bugs whose harm does not match what FCatch automatically checks in Section 4.3.3. (3) Custom synchronization. Like all techniques that identify custom

Table 1. FCatch Benchmarks.

App.	Version	Workload	Bench. Bugs
CA	1.1.12	Startup + AntiEntropy (AE)	CA1 [3], CA2 [4]
HB	0.96.0	Startup + HMasterRestart	HB1 [5]
HB	0.90.1	Startup	HB2 [1]
MR	0.23.1	Startup + WordCount(WC)	MR1 [2]
MR	2.1.1	Startup + WordCount(WC)	MR2 [6]
ZK	3.4.5	Startup	ZK [7]

synchronization [15, 52, 55], FCatch cannot guarantee to accurately identify all fault-tolerance mechanisms or synchronization loops. For example, FCatch does not consider sanity checks involving multiple variables or time-out mechanisms enforced by monitoring threads. (4) `RDTSCP` time stamp may cause inaccurate write-read and signal-wait pairing in FCatch. In practice, replay researchers have found it sufficiently accurate [43], which matches our experience. (5) FCatch selectively traces heap accesses for scalability concerns. Although neither sound nor complete, FCatch provides a good trade off in effectively detecting real-world TOF bugs with good accuracy in real-world large distributed systems, as shown by our evaluation.

7 Methodology

Benchmark software We evaluate FCatch on six common workloads in four widely used open-source distributed systems (Table 1), MapReduce distributed computing framework (MR); Cassandra distributed key-value stores (CA); HBase distributed key-value stores (HB); ZooKeeper distributed synchronization service (ZK), with 172K – 1,651K lines of code.

Benchmark bugs We obtain these workloads from TaxDC benchmark suite [37]: real-world users reported 7 TOF bugs, listed in Table 1, that could be triggered when specially-timed faults occur during these workloads. As the upper half of Table 2 shows, these 7 bugs cover both types of TOF bugs, a variety of contention resources like heap objects, files, and more, and a variety of failure symptoms, including system hangs, data losses, and job/system failures. These failures are all non-recoverable by these distributed systems — users/administrators have to re-submit the job or restart the system.

Evaluation Workload The 6 workloads under evaluation are all common ones, as shown in Table 1. Note that, the systems can tolerate *most* of the faults that occur while running these workloads. Our evaluation will show that these workloads can correctly finish even with hundreds of times' tries of randomly injected faults. FCatch detects TOF bugs by monitoring **correct** runs.

As discussed earlier, to detect Crash-Recovery TOF bugs, we randomly inject a node crash to trace recovery operations. To measure how sensitive FCatch is towards the time of fault injection, we did a sensitivity study. That is, for every benchmark, we apply FCatch three times, with faults injected at three different execution moments — near the beginning, in the middle, and near the end of the execution. We then check whether FCatch produces different bug reports among three different bug-detection attempts.

Experiment setting We run each benchmark in one virtual machine with VirtualBox 4.3.36. We use `VBoxManage` commands to take snapshot and restore each VM. Both host machine and guest OS in VM use Ubuntu 14.04 and JVM v1.7. Host machine has Intel® Xeon® CPU E5-2620 and 96GB of RAM. All trace analysis and pruning are on host machine.

Table 2. TOF bugs found by FCatch. Res.: resources in contention; H: heap object; ZK: ZKnode; GF: global file; LF: local file. More details can be found at https://github.com/haopeng-liu/TOF-bugs

ID	Operations	Res.	Symptom
	Benchmark Crash-Regular TOF bugs		
CA1	Signal vs Wait	H	AE hangs @ Snapshot
CA2	Signal vs Wait	H	AE hangs @ Mtree compare
HB1	Write vs Loop	H	HMaster hangs @ MetaOpen (Fig.6)
	Benchmark Crash-Recovery TOF bugs		
HB2	Create vs Create	ZK	Data loss as Get lock fail
MR1	Write vs Read	H	Task recovery hangs (Fig. 1)
MR2	Delete vs Open	GF	AM restart fails as Dir. deleted
ZK	Write vs Read	LF	Restart fails
	Non-Benchmark Crash-Regular TOF bugs		
CA3	Write vs Loop	H	AE hangs @ Mtree repair
HB3	Signal vs Wait	H	HMaster hangs @ ROOT open
HB4	Write vs Loop	H	HMaster hangs @ ROOT open
MR3	Signal vs Wait	H	Hangs @ Any RPC call
	Non-Benchmark Crash-Recovery TOF bugs		
HB5	Delete vs Read	ZK	Data loss as HLog skipped
HB6	Delete vs Read	ZK	Data loss as HLog dir. skipped
MR4	Write vs Read	H	Task recovery killed
MR5	Create vs Exists	GF	AM restart fails as Flag-file exists

Evaluation metrics We will evaluate FCatch from several aspects: the coverage and accuracy of its bug detection, the overhead of its run-time and static analysis, and how it compares with alternative designs and random fault injection.

All the performance numbers are based on an average of 3 runs during **correct** runs, with the baseline performance measured with **no** sleep inserted.

8 Experimental Results

8.1 Bug detection results

FCatch issues 31 TOF bug reports by analyzing correct runs:

- 8 are severe bugs explaining all the 7 benchmark issues;
- 8 are severe bugs that we were unaware of[4];
- 6 are false positives that cause benign exceptions;
- 9 are false positives that are incorrectly reported.

FCatch is effective in accurately predicting FCatch bugs.

8.1.1 Crash-Regular TOF bugs detecting

As shown in Table 3, FCatch not only correctly predicts three crash-regular benchmark TOF bugs (CA1, CA2, and HB1), but also finds 4 non-benchmark harmful TOF bugs (CA3, HB3, HB4, and MR3 in Table 2), with only 5 false positives across all benchmarks.

These four bugs are severe, as shown in Table 2. For example, MR3 affects *all* RPC calling. It can be triggered by either a crash of

[4]These 8 bugs are not part of existing benchmark suites [21, 37]. After carefully checking software change logs, we later found that (1) 4 bugs have been acknowledged and fixed by developers, and (2) the other 4 have never been reported, but have disappeared in latest versions due to software functionality changes. More details about these bugs, including their reproduction scripts, are available at https://github.com/haopeng-liu/TOF-bugs.

Table 3. FCatch bug detection results. *: same bug. Old: benchmark bugs; - : benchmark bug is not of this category; Exp.: false positives that throw exceptions.

	Crash–Regular				Crash–Recovery			
	Bugs		False		Bugs		False	
	Old	New	Exp.		Old	New	Exp.	
CA1&2	2	1	0	0	-	0	0	2
HB1	1	0	0	3	-	0	4	2
HB2	-	2	2	0	1	2	0	0
MR1	-	1*	0	0	1	1	0	0
MR2	-	1*	0	0	2	1	0	0
ZK	-	0	0	0	1	0	0	2
Total	3	4	2	3	5	4	4	6

the RPC caller or a drop of an RPC-return message, and causes the RPC caller to hang forever. As another example, HB3 and HB4 can both cause the whole HBase system to hang. We have triggered all these bugs.

False positives The 5 false positive crash-regular TOF bug reports fall into three categories. Two of them, both in HB2, indeed cause HMaster to hang. However, this is an expected behavior: when all region servers crash during registration, HMaster is expected to stuck in a loop waiting for at least one region server to come alive. One false positive in HB1 is caused by incorrectly identified custom loop-signal operation. The remaining two false positives in HB1 are caused by un-recognized time-out mechanisms: some components in the distributed system did hang, but another watcher-component in the system discovers and terminates the hang.

8.1.2 Crash-Recovery TOF bugs detecting

As shown in Table 3, FCatch not only correctly predicts four crash-recovery benchmark TOF bugs, including two TOF bugs that represent two ways to trigger MR2, but also finds four harmful TOF bugs we were unaware of, with two in MR and two in HB.

These 4 bugs are severe, as shown in Table 2, and have all been triggered by us. For example, HB5 and HB6 can cause HLogs or HLog directories to be skipped during HLog replication, which could then cause silent data loss in HBase.

False positives The 10 false positives of FCatch crash-recovery bug detection falls into two categories. For 4 of them (4 in HB-1), the reported {W, R} pairs can indeed cause exceptions, like `ZKNodeNotExistException` and `FileAlreadyExistsException`, when a fault happens at the reported moment. However, these exceptions are well handled. For example, when HBase' recovery routine finds that a temporary file F it plans to create already exists (i.e., left by the crashing node), it simply creates a temporary file F' with a different name. For the other 6 false positives, although the reported fault timing can indeed cause a recovery operation to consume different content from a shared resource from what it used to consume in correct runs, the difference is valid and does not lead to exceptions or failures.

Crash point sensitivity To detect crash-regular TOF bugs, FCatch only analyzes a fault-free run; to detect crash-recovery TOF bugs, FCatch analyzes not only a fault-free run but also a faulty run.

Table 4. FCatch performance. (Baseline: software running w/o any instrumentation. NF: fault-free run; F: faulty run; Reg: analysis to detect crash-regular bugs; Rec: analysis to detect crash-recovery bugs. Units: seconds. Slowdown = $\frac{Tracing+Analysis}{Baseline(NF)}$)

	Baseline		Tracing		Analysis		Overall	
	NF	F	NF	F	Reg	Rec	Time	Slowdown
CA1&2	17.5	25.4	36	59	9	104	208	11.9X
HB1	40.6	84.3	191	233	43	124	592	14.5X
HB2	14.5	54.4	67	125	10	19	221	15.2X
MR1	64.3	78.9	158	180	24	15	376	5.9X
MR2	68.4	114	274	284	21	93	672	9.8X
ZK	10.3	14.7	12	31	6	8	58	5.6X

Therefore, we evaluate whether the bug-detection results of FCatch is sensitive to where the fault occurs during the correct faulty run. Specifically, for each benchmark, we tried random fault-injection during three phases of the execution to collect the correct faulty-run traces: at the beginning (all other results in this section comes from this setting), in the middle, and near the end.

Our evaluation found that, the crash-recovery TOF bugs detected by FCatch are **not** sensitive to the fault location. No matter the fault occurs at the beginning or in the middle, FCatch reports exactly the same 16 severe bugs listed in Table 3. The only difference among the three sets of experiments is that, while injecting faults near the end of the execution, FCatch misses 2 bug reports, 1 in MR and 1 in ZK (FCatch actually misses 2 bug reports in MR1 benchmark, but one of them is reported by MR2 benchmark). These two bug reports are missed because when a node crashes, it has finished most of the work and hence the corresponding recovery routine conducted fewer operations comparing with other settings.

This evaluation shows that FCatch did not predict those bugs in Table 3 by luck and is much less sensitive to the timing of fault than traditional fault-injection testing.

8.2 Performance evaluation

Table 4 presents performance breakdowns of FCatch bug detection. Overall, FCatch is fast enough for in-house testing. Comparing with running a benchmark **once** with **no** fault **no** instrumentation **no** sleep injected, FCatch finishes its TOF bug detection with 5.6x – 15.2x slowdown. In fact, FCatch only imposes 3.8x – 8.2x slowdown for correct faulty runs.

For most benchmarks, FCatch spent more time in tracing than doing trace analysis. A key factor for the tracing overhead is that we use Javassist, a dynamic instrumentation tool, to inject tracing code. The performance would be better if we use a static code transformation tool. Trace analysis is mostly fast. The most time-consuming part in trace analysis to check data-dependency based fault tolerance mechanism.

The size of the traces produced by FCatch ranges from 9MB (ZK) to about 450 MB (MR). The trace size does not directly affect FCatch trace analysis time, as larger traces do not always contain more conflicting operations.

The performance of FCatch greatly benefits from its selective, instead of exhaustive, heap-access monitoring (Section 3.2). To demonstrate that, we also measured the run-time overhead of tracing *all* heap accesses. Our evaluation shows that CA benchmarks simply cannot finish, as the huge tracing overhead affects gossip

Table 5. # false positives pruned by different analysis (Dependence: dependence-based tolerance analysis)

	Crash–Regular		Crash–Recovery	
	Loop TimeOut	Wait TimeOut	Dependence	Impact
CA1&2	0	1	54	194
HB1	3	7	97	115
HB2	0	2	14	40
MR1	0	1	2	4
MR2	0	2	19	38
ZK	2	2	35	40

protocols and causes nodes to treat neighbors as dead; in MR benchmarks, mappers finish only 20% and reducers 0% after 20 minutes of execution (original FCatch finishes everything in 1–2 minutes, as shown in Table 4); HB benchmarks finish with 51X slowdowns; ZK starts up successfully but incurs 12X slowdowns. Overall, tracing all heap accesses is impractical for applying FCatch to large real-world distributed systems.

8.3 Random Crash Injection

We also compare FCatch with random fault injection. Specifically, our fault-injection script runs each workload for 400 times, and in every run picks a random time point to crash a node. We then analyze the log to see if any bugs is exposed.

Performance As shown in Table 4, comparing with one fault-injection run, FCatch bug detection incurs about 4x slowdown to HB2, MR, and ZK, and 7x–8x slowdown to CA and HB1.

Bug detection 400 runs of random fault-injection is much less effective than 1 round of FCatch— FCatch finds 16 unique true bugs, yet random fault-injection finds only 2. One is benchmark CA2, exposed 10 times in 400 runs. CA2 can be triggered if a node crashes in the middle of taking snapshot. Since taking snapshot takes time, CA2 has a decent probability (~2.5%) to be triggered by random fault injection. The other bug is in MR, a false negative of FCatch. It causes the system to hang while AM waits for a heartbeat change yet the node that can make the change crashes. FCatch missed this bug, because it traces only heap accesses inside event/message/RPC handlers. Unfortunately, the W operation in this bug is not inside such handlers, and instead has data dependence on the return value of such a handler.

We did not try other fault-injection schemes, as it is difficult to do better than random injection without sophisticated analysis or modeling. One possible alternative is to only inject faults around message sending and receiving. However, since many TOF bugs (7 out of 16 TOF bugs reported by FCatch) require faults between specific memory/file accesses to manifest, injecting faults around message operations has no chances to expose them. Furthermore, many of our benchmark executions contain more than 1000 messages, and hence still leave a huge search space for fault injection around message operations.

8.4 Pruning & Triggering

False positive pruning Table 5 shows that the fault-tolerance analysis (i.e., time-out analysis for crash-regular bugs in Section 4.2.2 and dependence analysis for crash-recovery bugs in Section 4.3.2) and the impact estimation (Section 4.3.3) greatly improve FCatch accuracy. Without them, the number of false positives will increase by about 5X for crash-regular bugs and about 40X for crash-recovery bugs.

Table 5 shows fewer crash-regular false positives pruned than those crash-recovery false positives pruned. A main reason is that before applying any static pruning analysis, FCatch already quires the conflicting operations W and R to satisfy blocking happens-before relationship in crash-regular TOF bug detection, and consequently leaves fewer false positives to be pruned by static analysis shown in Table 5.

Naively reporting all synchronization loops or file operations without existence checks as TOF bugs are naturally very inaccurate. We skip the numbers for space constraints.

Of course, our analysis could prune out true bugs. We randomly checked 10 pruned bug reports from each category. Among them, all are indeed false-positives.

Triggering The triggering component of FCatch, as discussed in Section 5, can easily trigger each reported bug and tell whether the bug report is false positive or not. While trying three different types of faults to trigger crash-regular TOF bug reports, we have sometimes experienced different impact of different faults. For example, the HB1 benchmark bug can only be triggered by a node-crash fault, but not by a kernel-level message drop fault, as the latter is handled by HBase through message resend. On the other hand, 2 out of the 3 crash-regular bugs from CA can be triggered by kernel-level message drops, but not node crashed, as the latter is handled by crash-recovery mechanisms in CA. For all other true crash-regular bugs in MR and HB2, they can be triggered by both node crashes and message drops.

9 Related Work

Distributed system fault injection framework In addition to the work discussed in Section 1, several techniques were proposed to help fault-injection testing in distributed systems. They provide domain specific languages that allow developers to specify fault-injection sequences, scenarios, and consistency specifications [20, 29, 38]. Although helpful, they do not solve the fundamental problem of the huge state space of distributed systems. For example, Fate [20] still has to use brute-force search to inject faults at every I/O operation point in program.

Model checking Distributed system model checkers intercept non-deterministic distributed events and permutes their ordering [23, 30, 36, 51, 56]. With different target from FCatch, these model checkers could find bugs that FCatch cannot. At the same time, they all suffer from state space explosion problems for real-world large distributed systems, as discussed in Section 1. They all only inject node crashes with coarse granularity, such as at the boundary of thread creation or message sending, which is insufficient to trigger many TOF bugs like MR2 and ZK in our benchmark suite. Since these model checkers do not only focus on fault-related bugs, they have to explore a lot of fault-unrelated states and struggle at scale to the whole distributed systems.

Verification Recent work proposed to build verifiable distributed system protocols [18, 25, 54] using proving frameworks like Coq [14] and TLA [34]. This direction is inspiring, but the state of the art is still far away from building verifiable systems that can match the performance and scale of existing systems. FCatch focuses on predicting TOF bugs in existing distributed systems and is orthogonal to this line of research.

Crash consistency Crash-consistency problem has been well studied for file systems and persistent memory systems. Symbolic execution and model checking were used to find consistency bugs in local file systems [32, 57, 58]. Recent work uses protocol-aware or domain knowledge to conduct fault injections to find out how distributed storage systems can tolerate correlated data loss [9, 19] and how OLTP databases can handle power faults [60]. Recent research proposes consistency models [46], consistency-enforcement techniques [12, 31, 53], and testing framework [35] that address consistency-problems unique to persistent memory platforms. These problems and techniques are all different from FCatch.

Error-handling bug detection Much previous work studied how to detect incorrectly-implemented exception handler or missing exception handlers [22, 48, 49, 59]. They do not look at the timing problem of faults and solve an orthogonal problem from FCatch.

Concurrency-bug detection Concurrency-bug detection for single-machine multi-threaded software has been studied for decades. Recently, tools have also been built for race detection in event-driven systems [26, 41, 47], message race detection in distributed systems [39], and serializability violation detection in applications that use eventually consistent data stores [13].

TOF bugs share some similarities with traditional concurrency bugs. For example, they are both triggered by special timing, and their root causes are both related to conflicting accesses to shared resources. Particularly, FCatch leverages traditional happens-before model and causality analysis in distributed systems [39, 40] and event-driven systems [26, 41, 47].

However, there are also many fundamental differences between TOF bugs and traditional concurrency bugs, which lead to completely different bug-detection designs. Most importantly, happens-before relationship is no longer a golden rule to detect TOF bugs. Many traditional concurrency-bug detection looks for conflicting operations that have no happens-before relationship with each other [44, 50] (i.e., data races). In the context of TOF bugs, conflicting operations **never** race with each other in traditional terms. Specifically, conflicting operations in crash-regular TOF bugs have happens-before relationship with each other; conflicting operations in crash-recovery TOF bugs often do not appear in one correct run and hence cannot be modeled by traditional happens-before relationship. Due to these differences, traditional concurrency-bug detection techniques can**not** find TOF bugs.

There are also other detailed differences between traditional concurrency bugs and TOF bugs. Traditional concurrency-bug detection only needs to observe fault-free runs, while TOF bug detection may require observing faulty runs. Traditional concurrency-bug detection cares about contention on all heap/global objects, while heap content of a crashed node does not matter for TOF bugs and content of persistent storage matters a lot for TOF bugs. Finally, synchronization mechanisms, like locks and conditional variables, are crucial to traditional concurrency-bug detection, but play no roles in TOF bugs.

10 Conclusions

Time-of-fault bugs are a unique type of bugs for distributed systems. They all severely hurt the system availability, because they do not have the fault-tolerance safety net as many other bugs. Different from traditional fault-injection approach, our work explores a new perspective to model and detect TOF bugs. Our evaluation shows that FCatch can indeed effectively detect TOF bugs with high accuracy and decent overhead.

Acknowledgments

We would like to thank Satish Narayanasamy, our shepherd, and the anonymous reviewers for their insightful feedback and comments. This research is supported by NSF (grants CNS-1563956, IIS-1546543, CNS-1514256, CCF-1514189, CCF-1439091) and generous supports from Huawei, Google Faculty Research Award, and CERES Center for Unstoppable Computing.

References

[1] Hbase-3596. https://issues.apache.org/jira/browse/HBASE-3596, 2011.
[2] Mapreduce-3858. https://issues.apache.org/jira/browse/MAPREDUCE-3858, 2012.
[3] Cassandra-5393. https://issues.apache.org/jira/browse/CASSANDRA-5393, 2013.
[4] Cassandra-6415. https://issues.apache.org/jira/browse/CASSANDRA-6415, 2013.
[5] Hbase-10090. https://issues.apache.org/jira/browse/HBASE-10090, 2013.
[6] Mapreduce-5476. https://issues.apache.org/jira/browse/MAPREDUCE-5476, 2013.
[7] Zookeeper-1653. https://issues.apache.org/jira/browse/ZOOKEEPER-1653, 2013.
[8] Java platform standard edition 7 documentation. https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode(), 2017.
[9] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Correlated crash vulnerabilities. In *OSDI*, 2016.
[10] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. Automating failure testing research at internet scale. In *SoCC*, 2016.
[11] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven Fault Injection. In *SIGMOD*, 2015.
[12] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *OOPSLA*, 2016.
[13] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In *POPL*, 2017.
[14] Pierre Castéran and Yves Bertot. Interactive theorem proving and program development. coq'art: The calculus of inductive constructions., 2004.
[15] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jPredictor: a predictive runtime analysis tool for java. In *ICSE*, 2008.
[16] Datapath.io. Recent aws outage and how you could have avoided downtime. https://medium.com/@datapath$_{i}$o/recent-aws-outage-and-how-you-could-have-avoided-downtime-7d9d9443d776, 2017.
[17] Jeff Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 2009.
[18] Pantazis Deligiannis, Alastair F Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. In *PLDI*, 2015.
[19] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *FAST*, 2017.
[20] 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.
[21] 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.
[22] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: error handling is occasionally correct. In *FAST*, 2008.
[23] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP*, 2011.
[24] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Peter Bodik, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure recovery: When the cure is worse than the disease. In *HotOS*, 2013.
[25] 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.
[26] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L Pereira, Gilles A Pokam, Peter M Chen, and Jason Flinn. Race detection for event-driven

mobile applications. In *PLDI*, 2014.
[27] IBM. Main page - walawiki. http://wala.sourceforge.net/wiki/index.php/Mainpage.
[28] jboss javassist. Javassist. http://jboss-javassist.github.io/javassist/.
[29] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. Setsudō: perturbation-based testing framework for scalable distributed systems. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, 2013.
[30] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.
[31] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. In *ASPLOS*, 2016.
[32] Eric Koskinen and Junfeng Yang. Reducing crash recoverability to reachability. In *POPL*, 2016.
[33] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
[34] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.
[35] Philip Lantz, Dulloor Subramanya Rao, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *ATC*, 2014.
[36] 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.
[37] 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.
[38] Kaituo Li, Pallavi Joshi, Aarti Gupta, and Malay K Ganai. Reprolite: A lightweight tool to quickly reproduce hard system bugs. In *SoCC*, 2014.
[39] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. In *ASPLOS*, 2017.
[40] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *SOSP*, 2015.
[41] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *PLDI*, 2014.
[42] IHS Markit. Businesses losing $700 billion a year to it downtime, says ihs. http://news.ihsmarkit.com/press-release/technology/businesses-losing-700-billion-year-it-downtime-says-ihs, 2016.
[43] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. In *ASPLOS*, 2017.
[44] Robert H. B. Netzer and Barton P. Miller. Improving The Accuracy of Data Race Detection. In *PPoPP*, 1991.
[45] Oracle. Virtualbox âĂŞ oracle vm virtualbox. https://www.virtualbox.org/wiki/VirtualBox.
[46] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *ISCA*, 2014.
[47] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *PLDI*, 2012.
[48] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *PLDI*, 2009.
[49] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, and Gilles Muller. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *DSN*, 2013.
[50] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM TOCS*, 1997.
[51] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *SSV*, 2010.
[52] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic Recognition of Synchronization Operations for Improved Data Race Detection. In *ISSTA*, 2008.
[53] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.
[54] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, 2015.
[55] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad Hoc Synchronization Considered Harmful. In *OSDI*, 2010.
[56] 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.
[57] Junfeng Yang, Can Sar, and Dawson Engler. Explode: a lightweight, general system for finding serious storage system errors. In *OSDI*, 2006.
[58] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *OSDI*, 2004.
[59] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*, 2014.
[60] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *OSDI*, 2014.