Minimizing Faulty Executions of Distributed Systems

Colin Scott[⋆] Aurojit Panda[⋆] Vjekoslav Brajkovic[◇] George Necula[⋆]
Arvind Krishnamurthy[†] Scott Shenker[⋆◇]
[⋆]*UC Berkeley* [◇]*ICSI* [†]*University of Washington*

Abstract

When troubleshooting buggy executions of distributed systems, developers typically start by manually separating out events that are responsible for triggering the bug (signal) from those that are extraneous (noise). We present DEMi, a tool for automatically performing this minimization. We apply DEMi to buggy executions of two very different distributed systems, Raft and Spark, and find that it produces minimized executions that are between 1X and 4.6X the size of optimal executions.

1 Introduction

Even simple code can contain bugs (e.g., crashes due to unexpected input). But the developers of distributed systems face additional challenges, such as concurrency, asynchrony, and partial failure, which require them to consider all possible ways that non-determinism might manifest itself. Since the number of event orderings a distributed system may encounter grows exponentially with the number of events, bugs are commonplace.

Software developers discover bugs in several ways. Most commonly, they find them through unit and integration tests. These tests are ubiquitous, but they are limited to cases that developers anticipate themselves. To uncover unanticipated cases, semi-automated testing techniques such as fuzzing (where sequences of message deliveries, failures, etc. are injected into the system) are effective. Finally, despite pre-release testing, bugs may turn up once the code is deployed in production.

The last two means of bug discovery present a significant challenge to developers: the system can run for long periods before problems manifest themselves. The resulting executions can contain a large number of events, most of which are not relevant to triggering the bug. Understanding how a trace containing thousands of concurrent events lead the system to an unsafe state requires significant expertise, time,[1] and luck.

Faulty execution traces can be made easier to understand if they are first *minimized*, so that only events that are relevant to triggering the bug remain. In fact, developers often start troubleshooting by manually performing this minimization. Since developer time is typically much more costly than machine time, automated minimization tools for *sequential* test cases [24, 86, 94] have already proven themselves valuable, and are routinely applied to bug reports for software projects such as Firefox [1], LLVM [7], and GCC [6].

In this paper we address the problem of automatically minimizing executions of distributed systems. We focus on executions generated by fuzz testing, but we also illustrate how one might minimize production traces.

Distributed executions have two distinguishing features. Most importantly, input events (e.g., failures) are *interleaved* with internal events (e.g., intra-process message deliveries) of concurrent processes. Minimization algorithms must therefore consider both which input events and which (of the exponentially many) event schedules are likely to still trigger the bug. Our main contribution (discussed in section 3) is a set of techniques for searching through the space of event schedules in a timely manner; these techniques are inspired by our understanding of how practical systems behave.

Distributed systems also frequently exhibit non-determinism (e.g., since they make extensive use of timers to detect failures), complicating replay. We address this challenge (as we discuss in section 4) by instrumenting the Akka actor system framework [2] to gain nearly perfect control over when events occur.

With the exception of our prior work [70], we are unaware of any other tool that solves this problem without needing to analyze the code. Our prior work targeted a specific distributed system (SDN controllers), and focused on minimizing input events given limited control over the execution [70]. Here we target a broader range of systems, define the general problem of execution minimization, exercise significantly greater control, and systematically explore the state space. We also articulate new minimization strategies that quickly reduce input events, internal events, and message contents.

Our tool, Distributed Execution Minimizer (DEMi), is implemented in ∼14,000 lines of Scala. We have applied DEMi to akka-raft [3], an open source Raft consensus implementation, and Apache Spark [90], a widely used data analytics framework. Across 10 known and discovered bugs, DEMi produces executions that are within a factor of 1X to 4.6X (1.6X median) the size of the smallest possible bug-triggering execution, and between

[1] Developers spend a significant portion of their time debugging (49% of their time according to one study [52]), especially when the bugs involve concurrency (70% of reported concurrency bugs in [37] took days to months to fix).

1X and 16X (4X median) smaller than the executions produced by the previous state-of-the-art blackbox technique [70]. The results we find for these two very different systems leave us optimistic that these techniques, along with adequate visibility into events (either through a framework like Akka, or through custom monitoring), can be applied successfully to a wider range of systems.

2 Problem Statement

We start by introducing a model of distributed systems as groundwork for defining our goals. As we discuss further in §4.2, we believe this model is general enough to capture the behavior of many practical systems.

2.1 System Model

Following [33], we model a distributed system as a collection of N single-threaded processes communicating through messages. Each process p has unbounded memory, and behaves deterministically according to a transition function of its current state and the messages it receives. The overall system S is defined by the transition function and initial configuration for each process.

Processes communicate by sending messages over a network. A message is a pair (p,m), where p is the identity of the destination process, and m is the message value. The network maintains a buffer of pending messages that have been sent but not yet delivered. Timers are modeled as messages a process can request to be delivered to itself at a specified later point in the execution.

A *configuration* of the system consists of the internal state of each process and the contents of the network's buffer. Initially the network buffer is empty.

An *event* moves the system from one configuration to another. Events can be one of two kinds. *Internal events* take place by removing a message m from the network's buffer and delivering it to the destination p. Then, depending on m and p's internal state, p enters a new internal state determined by its transition function, and sends a finite set of messages to other processes. Since processes are deterministic, internal transitions are completely determined by the contents of m and p's state.

Events can also be *external*. The three external events we consider are: process starts, which create a new process; forced restarts (crash-recoveries), which force a process to its initial state (though it may maintain non-volatile state); and external message sends (p,m), which insert a message sent from outside the system into the network buffer (which may be delivered later as an internal event). We do not need to explicitly model fail-stop failures, since these are equivalent to permanently partitioning a process from all other processes.

A *schedule* is a finite sequence τ of events (both external and internal) that can be applied, in turn, starting from an initial configuration. Applying each event in the schedule results in an *execution*. We say that a schedule 'contains' a sequence of external events $E = [e_1, e_2, \ldots, e_n]$ if it includes only those external events (and no other external events) in the given order.

2.2 Testing

An *invariant* is a predicate P (a safety condition) over the internal state of all processes at a particular configuration C. We say that configuration C violates the invariant if $P(C)$ is false, denoted $\overline{P}(C)$.

A *test orchestrator* generates sequences of external events $E = [e_1, e_2, \ldots, e_n]$, executes them along with some (arbitrary) schedule of internal events, and checks whether any invariants were violated during the execution. The test orchestrator records the external events it injected, the violation it found, and the interleavings of internal events that appeared during the execution.

2.3 Problem Definition

We are given a schedule τ injected by a test orchestrator,[2] along with a specific invariant violation $\overline{P}$ observed at the end of the test orchestrator's execution.

Our main goal is to find a schedule containing a small sequence of external (input) events that reproduces the violation $\overline{P}$. Formally, we define a minimal causal sequence (MCS) to be a subsequence of external events $E' \sqsubseteq E$ such that there exists a schedule containing E' that produces $\overline{P}$, but if we were to remove any single external event e from E', there would not exist any schedules shorter[3] than τ containing $E' - e$ that produce $\overline{P}$.[4]

We start by minimizing external (input) events because they are the first level of abstraction that developers reason about. Occasionally, developers can understand the root cause simply by examining the external events.

For more difficult bugs, developers typically step through the internal events of the execution to understand more precisely how the system arrived at the unsafe state. To help with these cases, we turn to minimizing internal events after the external events have been minimized. At this stage we fix the external events and search for smaller schedules that still triggers the invariant violation, for example, by keeping some messages pending rather than delivering them. Lastly, we seek to minimize the contents (e.g. data payloads) of external messages.

Note that we do not focus on bugs involving only sequential computation (e.g. incorrect handling of unex-

[2] We explain how we obtain these schedules in §4.

[3] We limit the number of internal events to ensure that the search space is finite; any asynchronous distributed system that requires delivery acknowledgment is not guaranteed to stop sending messages [8], essentially because nodes cannot distinguish between crashes of their peers and indefinite message delays.

[4] It might be possible to reproduce $\overline{P}$ by removing multiple events from E', but checking all combinations is tantamount to enumerating its powerset. Following [94], we only require a 1-minimal subsequence E' instead of a globally minimal subsequence.

pected input), performance, or human misconfiguration. Those three bug types are more common than our focus: concurrency bugs. We target concurrency bugs because they are the most complex (correspondingly, they take considerably more time to debug [37]), and because mature debugging tools already exist for sequential code.

With a minimized execution in hand, the developer begins debugging. Echoing the benefits of sequential test case minimization, we claim that the greatly reduced size of the trace makes it easier to understand which code path contains the underlying bug, allowing the developer to focus on fixing the problematic code itself.

3 Approach

Conceptually, one could find MCSes by enumerating and executing every possible (valid, bounded) schedule containing the given external events. The globally minimal MCS would then be the shortest sequence containing the fewest external events that causes the safety violation. Unfortunately, the space of all schedules is exponentially large, so executing all possible schedules is not feasible. This leads us to our key challenge:

> *How can we maximize reduction of trace size within bounded time?*

To find MCSes in reasonable time, we split schedule exploration into two parts. We start by using delta debugging [94] (shown in Appendix A), a minimization algorithm similar to binary search, to prune extraneous external events. Delta debugging works by picking subsequences of external events, and checking whether it is possible to trigger the violation with just those external events starting from the initial configuration. We assume the user gives us a time budget, and we spread this budget evenly across each subsequence's exploration.

To check whether a particular subsequence of external events results in the safety violation, we need to explore the space of possible interleavings of internal events and external events. We use Dynamic Partial Order Reduction ('DPOR', shown in Appendix B) to prune this schedule space by eliminating equivalent schedules (i.e. schedules that differ only in the ordering of commutative events [34]). DPOR alone is insufficient though, since there are still exponentially many non-commutative schedules to explore. We therefore prioritize the order in which we explore the schedule space.

For any prioritization function we choose, an adversary could construct the program under test to behave in a way that prevents our prioritization from making any progress. In practice though, programmers do not construct adversarial programs, and test orchestrators do not construct adversarial inputs. We choose our prioritization order according to observations about how the programs we care about behave in practice.

Our central observation is that if one schedule triggers a violation, schedules that are similar in their causal structure should have a high probability of also triggering the violation. Translating this intuition into a prioritization function requires us to address our second challenge:

> *How can we reason about the similarity or dissimilarity of two different executions?*

We implement a hierarchy of *match* functions that tell us whether messages from the original execution correspond to the same logical message from the current execution. We start our exploration with a single, uniquely-defined schedule that closely resembles the original execution. If this schedule does not reproduce the violation, we begin exploring nearby schedules. We stop exploration once we have either successfully found a schedule resulting in the desired violation, or we have exhausted the time allocated for checking that subsequence.

External event minimization ends once the system has successfully explored all subsequences generated by delta debugging. Limiting schedule exploration to a fixed time budget allows minimization to finish in bounded time, albeit at the expense of completeness (i.e., we may not return a perfectly minimal event sequence).

To further minimize execution length, we continue to use the same schedule exploration procedure to minimize internal events once external event minimization has completed. Internal event minimization continues until no more events can be removed, or until the time budget for minimization as a whole is exhausted.

Thus, our strategy is to (i) pick subsequences with delta debugging, (ii) explore the execution of that subsequence with a modified version of DPOR, starting with a schedule that closely matches the original, and then by exploring nearby schedules, and (iii) once we have found a near-minimal MCS, we attempt to minimize the number of internal events. With this road map in mind, below we describe our minimization approach in greater detail.

3.1 Choosing Subsequences of External Events

We model the task of minimizing a sequence of external events E that causes an invariant violation as a function $ExtMin$ that repeatedly removes parts of E and invokes an oracle (defined in §3.2.1) to check whether the resulting subsequence, E', still triggers the violation. If E' triggers the violation, then we can assume that the parts of E removed to produce E' are not required for producing the violation and are thus not a part of the MCS.

$ExtMin$ can be trivially implemented by removing events one at a time from E, invoking the oracle at each iteration. However, this would require that we check $O(|E|)$ subsequences to determine whether each triggers the violation. Checking a subsequence is expensive,

since it may require exploring a large set of event schedules. We therefore apply delta debugging [93, 94], an algorithm similar to binary search, to achieve $O(log(|E|))$ average case runtime (worst case $O(|E|)$). The delta debugging algorithm we use is shown in Appendix A.

Efficient implementations of *ExtMin* should not waste time trying to execute invalid (non-sensical) external event subsequences. We maintain validity by ensuring that forced restarts are always preceded by a start event for that process, and by assuming that external messages are independent of each other, i.e., we do not currently support external messages that, when removed, cause some other external event to become invalid. One could support minimization of dependent external messages by either requiring the user to provide a grammar, or by employing the $O(|E|^2)$ version of delta debugging that considers complements [94].

3.2 Checking External Event Subsequences

Whenever delta debugging selects an external event sequence E', we need to check whether E' can result in the invariant violation. This requires that we enumerate and check all schedules that contain E' as a subsequence. Since the number of possible schedules is exponential in the number of events, pruning this schedule space is essential to finishing in a timely manner.

As others have observed [38], many events occurring in a schedule are *commutative*, i.e., the system arrives at the same configuration regardless of the order events are applied. For example, consider two events e_1 and e_2, where e_1 is a message from process a being delivered to process c, and e_2 is a message from process b being delivered to process d. Assume that both e_1 and e_2 are co-enabled, meaning they are both pending at the same time and can be executed in either order. Since the events affect a disjoint set of nodes (e_1 changes the state at c, while e_2 changes the state at d), executing e_1 before e_2 causes the system to arrive at the same state it would arrive at if we had instead executed e_2 before e_1. e_1 and e_2 are therefore commutative. This example illustrates a form of commutativity captured by the happens-before relation [51]: two message delivery events a and b are commutative if they are concurrent, i.e. $a \not\rightarrow b$ and $b \not\rightarrow a$, and they affect a disjoint set of nodes.

Partial order reduction (POR) [34,38] is a well-studied technique for pruning commutative schedules from the search space. In the above example, given two schedules that only differ in the order in which e_1 and e_2 appear, POR would only explore one schedule. Dynamic POR (DPOR) [34] is a refinement of POR (shown in Appendix B): at each step, it picks a pending message to deliver, dynamically computes which other pending events are not concurrent with the message it just delivered, and sets backtrack points for each of these, which it will later use (when exploring other non-equivalent schedules) to try delivering the pending messages in place of the message that was just delivered.

Even when using DPOR, the task of enumerating all possible schedules containing E as a subsequence remains intractable. Moreover, others have found that naïve DPOR gets stuck exploring a small portion of the schedule space because of its depth-first exploration order [57]. We address this problem in two ways: first, as mentioned before, we limit *ExtMin* so it spreads its fixed time budget roughly evenly across checking whether each particular subsequence of external events reproduces the invariant violation. It does this by restricting DPOR to exploring a fixed number of schedules before giving up and declaring that an external event sequence does not produce the violation. Second, to maximize the probability that invariant violations are discovered quickly while exploring a fixed number of schedules, we employ a set of schedule exploration strategies to guide DPOR's exploration, which we describe next.

3.2.1 Schedule Exploration Strategies

We guide schedule exploration by manipulating two degrees of freedom within DPOR: (i) we prescribe which pending events DPOR initially executes, and (ii) we prioritize the order backtrack points are explored in. In its original form, DPOR only performs depth-first search starting from an arbitrary initial schedule, because it was designed to be *stateless* so that it can run indefinitely in order to find as many bugs as possible. Unlike the traditional use case, our goal is to minimize a known bug in a timely manner. By keeping some state tracking the schedules we have already explored, we can pick backtrack points in a prioritized (rather than depth-first) order without exploring redundant schedules.

A scheduling strategy implements a backtrack prioritization order. Scheduling strategies return the first violation-reproducing schedule they find (if any) within their time budget. We design our key strategy (shown in Algorithm 1) with the following observations in mind:

Observation #1: Stay close to the original execution. The original schedule provides us with a 'guide' for how we can lead the program down a code path that makes progress towards entering the same unsafe state. By choosing modified schedules that have causal structures that are close to the original schedule, we should have high probability of retriggering the violation.

We realize this observation by starting our exploration with a single, uniquely defined schedule for each external event subsequence: deliver only messages whose source, destination, and contents 'match' (described in detail below) those in the original execution, in the exact same order that they appeared in the original execution. If an internal message from the original execution is not pend-

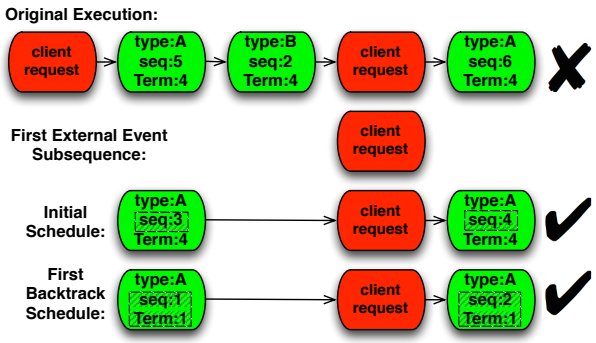

Figure 1: Example schedules. External message deliveries are shown in red, internal message deliveries in green. Pending messages, source addresses, and destination addresses are not shown. The 'B' message becomes absent when exploring the first subsequence of external events. We choose an initial schedule that is close to the original, except for the masked 'seq' field. The violation is not triggered after the initial schedule (depicted as ✔), so we next match messages by type, allowing us to deliver pending messages with smaller 'Term' numbers.

ing (i.e. sent previously by some actor) at the point that internal message should be delivered, we skip over it and move to the next message from the original execution. Similarly, we ignore any pending messages that do not match any events delivered in the original execution. In the case where multiple pending messages match, it does not matter which we choose (see Observation #2).

Matching Messages. A function *match* determines whether a pending message from a modified execution logically corresponds to a message delivered in the original execution. The simplest way to implement *match* is to check equality of the source, the destination, and all bytes of the message contents. Recall though that we are executing a *subsequence* of the original external events. In the modified execution the contents of many of the internal messages will likely change relative to message contents from the original execution. Consider, for example, sequence numbers that increment once for every message a process receives (shown as the 'seq' field in Figure 1). These differences in message contents prevent simple bitwise equality from finding many matches.

Observation #2: Data independence. Often, altered message contents such as differing sequence numbers do *not* affect the behavior of the program, at least with respect to whether the program will reach the unsafe state. Formally, this property is known as 'data-independence', meaning that the values of some message contents do not affect the system's control-flow [71, 82].

To leverage data independence, application developers can (optionally) supply us with a 'message fingerprint' function,[5] which given a message returns a string that depends on the relevant parts of the message, without considering fields that should be ignored when checking if two message instances from different executions refer to the same logical message. An example fingerprint function might ignore sequence numbers and authentication cookies, but concatenate the other fields of messages. Message fingerprints are useful both as a way of mitigating non-determinism, and as a way of reducing the number of schedules the scheduling strategy needs to explore (by drawing an equivalence relation between all schedules that only differ in their masked fields). We do not require strict data-independence in the formal sense [71]; the fields the user-defined fingerprint function masks over may in practice affect the control flow of the program, which is generally acceptable because we simply use this as a strategy to guide the choice of schedules, and can later fall back to exploring all schedules if we have enough remaining time budget.

We combine observations #1 and #2 to pick a single, unique schedule as the initial execution, defined by selecting pending events in the modified execution that *match* the original execution. This stage corresponds to the first two lines of TEST in Algorithm 1. We show an example initial schedule in Figure 1.

Challenge: history-dependent message contents. This initial schedule can be remarkably effective, as demonstrated by the fact that minimization often produces significant reduction even when we limit it to exploring this single schedule per external event subsequence. However, we find that without exploring additional schedules, the MCSes we find still contain extraneous events: when message contents depend on previous events, and the messages delivered in the original execution contained contents that depended on a large number of prior events, the initial schedule will remain inflated because it never includes "unexpected" pending messages that were not delivered in the original execution yet have contents that depend on fewer prior events.

To illustrate, let us consider two example faulty executions of the Raft consensus protocol. The first execution was problematic because all Raft messages contain logical clocks ("Term numbers") that indicate which epoch the messages belong to. The logical clocks are incremented every time there is a new leader election cycle. These logical clocks *cannot* be masked over by the message fingerprint, since they play an important role in determining the control flow of the program.

In the original faulty execution, the safety violation happened to occur at a point where logical clocks had high values, i.e. many leader election cycles had already taken place. We knew however that most of the leader election cycles in the beginning of the execution were not necessary to trigger the safety violation. Minimization restricted to only the initial schedule was *not* able to remove the earlier leader election cycles, though we

[5] It may be possible to extract message fingerprints automatically using program analysis or experimentation [77]. Nonetheless, manually defining fingerprints does not require much effort (see Table 4). Without a fingerprint function, we default to matching on message type (Observation #3).

Algorithm 1 Pseudocode for schedule exploration. TEST is invoked once per external event subsequence E'. We elide the details of DPOR for clarity (see Appendix B for a complete description). τ denotes the original schedule; b.counterpart denotes the message delivery that was delivered instead of b (variable m in the elif branch of STSSCHED); b.predecessors and b.successors denote the events occuring before and after b when b was set (τ''[0..i] and τ''[i+1...τ''.length] in STSSCHED).

```
backtracks ← {}
procedure TEST(E')
  STSSCHED(E',τ)
  if execution reproduced ✘: return ✘
  while ∃_{b∈backtracks.} b.type=b.counterpart.type ∧
      b.fingerprint ≠ b.counterpart.fingerprint ∧
      time budget for E' not yet expired do
    reinitialize system, remove b from backtracks
    prefix ← b.predecessors + [ b ]
    if prefix (or superstring) already executed:
      continue
    STSSCHED(E',prefix + b.successors)
    if execution reproduced ✘: return ✘
  return ✔
procedure STSSCHED(E',τ')
  τ'' ← τ'.remove {e | e is external and e ∉ E'}
  for i from 0 to τ''.length do
    if τ''[i] is external:
      inject τ''[i]
    elif ∃_{m∈pending.} m.fingerprint = τ''[i].fingerprint:
      deliver m, remove m from pending
      for m' ∈ pending do
        if ¬commute(m,m'):
          backtracks ← backtracks ∪ {m'}
```

would have been able to if we had instead delivered other pending messages with small term numbers.

The second execution was problematic because of *batching*. In Raft, the leader receives client commands, and after receiving each command, it replicates it to the other cluster members by sending them 'AppendEntries' messages. When the leader receives multiple client commands before it has successfully replicated them all, it batches them into a single AppendEntries message. Again, client commands cannot be masked over by the fingerprint function, and because AppendEntries are internal messages, we cannot shrink their contents.

We knew that the safety violation could be triggered with only one client command. Yet minimization restricted to only the initial schedule was unable to prune many client commands, because in the original faulty execution AppendEntries messages with large batch contents were delivered before pending AppendEntries messages with small batch contents.

These examples motivated our next observations:

Observation #3: Coarsen message matching. We would like to stay close to the original execution (per observation #1), yet the previous examples show that we should not restrict ourselves to schedules that only match according to the user-defined message fingerprints from the original execution. We can achieve both these goals by considering a more coarse-grained *match* function: the *type* of pending messages. By 'type', we mean the language-level type tag of the message object, which is available to the RPC layer at runtime.

We choose the next schedules to explore by looking for pending messages whose *types* (not contents) match those in the original execution, in the exact same order that they appeared in the original execution. We show an example in Figure 1, where any pending message of type 'A' with the same source and destination as the original messages would match. When searching for candidate schedules, if there are no pending messages that match the type of the message that was delivered at that step in the original execution, we skip to the next step. Similarly, we ignore any pending messages that do not match the corresponding type of the messages from the original execution. This leaves one remaining issue: how we handle cases where multiple pending messages match the corresponding original message's type.

Observation #4: Prioritize backtrack points that resolve match ambiguities. When there are multiple pending messages that match, we initially only pick one. DPOR (eventually) sets backtrack points for all other co-enabled dependent events (regardless of type or message contents). Of all these backtrack points, those that match the type of the corresponding message from the original trace should be most fruitful, because they keep the execution close to the causal structure of the original schedule except for small ambiguities in message contents.

We show the pseudocode implementing Observation #3 and Observation #4 as the while loop in Algorithm 1. Whenever we find a backtrack point (pending message) that matches the type but not the fingerprint of an original delivery event from τ, we replace the original delivery with the backtrack's pending message, and execute the events before and after the backtrack point as before.

Backtracking allow us to eventually explore all combinations of pending messages that match by type. Note here that we do not ignore the user-defined message fingerprint function: we only prioritize backtrack points for pending messages that have the same type *and* that differ in their message fingerprints.

Minimizing internal events. Once delta debugging over external events has completed, we attempt to further reduce the smallest reproducing schedule found so far. Here we apply delta debugging to internal events: for each subsequence of internal events chosen by delta debugging, we (i) mark those messages so that they are left pending and never delivered, and (ii) apply the same scheduling strategies described above for the remaining events to check whether the violation is still triggered.

Internal event minimization continues until there is no more minimization to be performed, or until the time budget for minimization as a whole is exhausted.

Observation #5: Shrink external message contents whenever possible. Our last observation is that the contents of external messages can affect execution length; because the test environment crafts these messages, it should minimize their contents whenever possible.

A prominent example is akka-raft's bootstrapping messages. akka-raft processes do not initially know which other processes are part of the cluster. They instead wait to receive an external bootstrapping message that informs them of the identities of all other processes. The contents of the bootstrapping messages (the processes in the cluster) determine *quorum size*: how many acknowledgments are needed to reach consensus, and hence how many messages need to be delivered. If the application developer provides us with a function for separating the components of such message contents, we can minimize their contents by iteratively removing elements, and checking to see if the violation is still triggerable until no single remaining element can be removed.

Recap. In summary, we first apply delta debugging (*ExtMin*) to prune external events. To check each external event subsequence chosen by delta debugging, we use a stateful version of DPOR. We first try exploring a uniquely defined schedule that closely matches the original execution. We leverage data independence by applying a user-defined message fingerprint function that masks over certain message contents. To overcome inflation due to history-dependent message contents, we explore subsequent schedules by choosing backtrack points according to a more coarse-grained match function: the types of messages. We spend the remaining time budget attempting to minimize internal events, and wherever possible, we seek to shrink external message contents.

3.3 Comparison to Prior Work

We made observations #1 and #2 in our prior work [70]. In this paper, we adapt observations #1 and #2 to determine the first schedule we explore for each external event subsequence (the first two lines of TEST). We refer to the scheduling strategy defined by these two observations as 'STSSched', named after the 'STS' system [70].

STSSched only prescribes a single schedule per external event subsequence chosen by delta debugging. In this work we systematically explore multiple schedules using the DPOR framework. We guide DPOR to explore schedules in a prioritized order based on similarity to the original execution (observations #3 and #4, shown as the while loop in TEST). We refer to the scheduling strategy used to prioritize subsequent schedules as 'TFB' (Type Fingerprints with Backtracks). We also minimize internal events, and shrink external message contents.

4 Systems Challenges

We implement our techniques in a publicly available tool we call DEMi (Distributed Execution Minimizer) [5]. DEMi is an extension to Akka [2], an actor framework for JVM-based languages. Actor frameworks closely match the system model in §2: actors are single-threaded entities that can only access local state and operate on messages received from the network one at a time. Upon receiving a message an actor performs computation, updates its local state and sends a finite set of messages to other actors before halting. Actors can be co-located on a single machine (though the actors are not aware of this fact) or distributed across multiple machines.

On a single machine Akka maintains a buffer of sent but not yet delivered messages, and a pool of message dispatch threads. Normally, Akka allows multiple actors to execute concurrently, and schedules message deliveries in a non-deterministic order. We use AspectJ [50], a mature interposition framework, to inject code into Akka that allows us to completely control when messages and timers are delivered to actors, thereby linearizing the sequence of events in an executing system. We currently run all actors on a single machine because this simplifies the design of DEMi, but minimization could also be distributed across multiple machines to improve scalability.

Our interposition lies above the network transport layer; DEMi makes delivery decisions for application-level (non-segmented) messages. If the application assumes ordering guarantees from the transport layer (e.g. TCP's FIFO delivery), DEMi adheres to these guarantees during testing and minimization to maintain soundness.

Fuzz testing with DEMi. We begin by using DEMi to generate faulty executions. Developers give DEMi a test configuration (we tabulate all programmer-provided specifications in Appendix C), which specifies an initial sequence of external events to inject before fuzzing, the types of external events to inject during fuzzing (along with probabilities to determine how often each event type is injected), the safety conditions to check (a user-defined predicate over the state of the actors), the scheduling constraints (e.g. TCP or UDP) DEMi should adhere to, the maximum execution steps to take, and optionally a message fingerprint function. If the application emits side-effects (e.g. by writing to disk), the test configuration specifies how to roll back side-effects (e.g. by deleting disk contents) at the end of each execution.

DEMi then repeatedly executes fuzz runs until it finds a safety violation. It starts by generating a sequence of random external events of the length specified by the configuration. DEMi then injects the initial set of external events specified by the developer, and then starts injecting external events from the random sequence. Developers can include special 'WaitCondition' markers in the initial set of events to execute, which cause DEMi

to pause external event injection, and deliver pending internal messages at random until a specified condition holds, at which point the system resumes injecting external events. DEMi periodically checks invariants by halting the execution and invoking the developer-supplied safety predicate over the current state of all actors. Execution proceeds until a predicate violation is found, the supplied bound on execution steps is exceeded, or there are no more external or internal events to execute.

Once it finds a faulty execution DEMi saves a user-defined fingerprint of the violation it found (a violation fingerprint might, for example, mark which process(es) exhibited the violation),[6] a totally ordered recording of all events it executed, and information about which messages were sent in response to which events. Users can then replay the execution exactly, or instruct DEMi to minimize the execution as described in §3.

Mitigating non-determinism. Processes may behave non-deterministically. A process is non-deterministic if the messages it emits (modulo fingerprints) are not uniquely determined by the prefix of messages we have delivered to it in the past starting from its initial state.

The main way we control non-determinism is by interposing on Akka's API calls, which operate at a high level and cover most sources of non-determinism. For example, Akka provides a timer API that obviates the need for developers to read directly from the system clock.

Applications may also contain sources of non-determinism outside of the Akka API. We discovered the sources of non-determinism described below through trial and error: when replaying unmodified test executions, the violation was sometimes not reproduced. In these cases we compared discrepancies between executions until we isolated their source and interposed on it.

akka-raft instrumentation. Within akka-raft, actors use a pseudo random number generator to choose when to start leader elections. Here we provided a seeded random number generator under the control of DEMi.

Spark instrumentation. Within Spark, the task scheduler chooses the first value from a hashmap in order to decide what tasks to schedule. The values of the hashmap are arbitrarily ordered, and the order changes from execution to execution. We needed to modify Spark to sort the values of the hash map before choosing an element.

Spark runs threads ('TaskRunners') that are outside the control of Akka. These send status update messages to other actors during their execution. The key challenge with threads outside Akka's control is that we do not know when the thread has started and stopped each step of its computation; when replaying, we do not know how long to wait until the TaskRunner either resends an expected message, or we declare that message as absent.

We add two interposition points to TaskRunners: the start of the TaskRunner's execution, and the end of the TaskRunner's execution. At the start of the TaskRunner's execution, we signal to DEMi the identity of the TaskRunner, and DEMi records a 'start atomic block' event for that TaskRunner. During replay, DEMi blocks until the corresponding 'end atomic block' event to ensure that the TaskRunner has finished sending messages. This approach works because TaskRunners in Spark have a simple control flow, and TaskRunners do not communicate via shared memory. Were this not the case, we would have needed to interpose on the JVM's thread scheduler.

Besides TaskRunner threads, the Spark driver also runs a bootstrapping thread that starts up actors and sends initialization messages. We mark all messages sent during the initialization phase as 'unignorable', and we have DEMi wait indefinitely for these messages to be sent during replay before proceeding. When waiting for an 'unignorable' message, it is possible that the only pending messages in the network are repeating timers. We prevent DEMi from delivering infinite loops of timers while it awaits by detecting timer cycles, and not delivering more timers until it delivers a non-cycle message.

Spark names some of the files it writes to disk using a timestamp read from the system clock. We hardcode a timestamp in these cases to make replay deterministic.

Akka changes. In a few places within the Akka framework, Akka assigns IDs using an incrementing counter. This can be problematic during minimization, since the counter value may change as we remove events, and the (non-fingerprinted) message contents in the modified execution may change. We fix this by computing IDs based on a hash of the current callstack, along with task IDs in case of ambiguous callstack hashes. We found this mechanism to be sufficient for our case studies.

Stop-gap: replaying multiple times. In cases where it is difficult to locate the cause of non-determinism, good reduction can often still be achieved simply by configuring DEMi to replay each schedule multiple times and checking if any of the attempts triggered the safety violation.

Blocking operations. Akka deviates from the computational model we defined in §2 in one remaining aspect: Akka allows actors to block on certain operations. For example, actors may block until they receive a response to their most recently sent message. To deal with these cases we inject AspectJ interposition on blocking operations (which Akka has a special marker for), and signal to DEMi that the actor it just delivered a message to will not become unblocked until we deliver the response message. DEMi then chooses another actor to deliver a message to, and marks the previous actor as blocked until

[6] Violation fingerprints should be specific enough to disambiguate different bugs found during minimization, but they do not need to be specific to the exact state the system at the time of the violation. Less specific violation fingerprints are often better, since they allow DEMi to find divergent code paths that lead to the same buggy behavior.

DEMi decides to deliver the response.

4.1 Limitations

Safety vs. liveness. We are primarily focused on safety violations, not liveness or performance bugs.

Non-Atomic External Events. DEMi currently waits for external events (e.g. crash-recoveries) to complete before proceeding. This may prevent it from finding bugs involving finer-grained event interleavings.

Limited scale. DEMi is currently tied to a single physical machine, which limits the scale of systems it can test (but not the bugs it can uncover, since actors are unaware of colocation). We do not believe this is fundamental.

Shared memory & disk. In some systems processes communicate by writing to shared memory or disk rather than sending messages over the network. Although we do not currently support it, if we took the effort to add interposition to the runtime system (as in [74]) we could treat writes to shared memory or disk in the same way we treat messages. More generally, adapting the basic DPOR algorithm to shared memory systems has been well studied [34, 85], and we could adopt these approaches.

Non-determinism. Mitigating non-determinism in akka-raft and Spark required effort on our part. We might have adopted deterministic replay systems [29, 36, 56, 92] to avoid manual instrumentation. We did not because we could not find a suitably supported record and replay system that operates at the right level of abstraction for actor systems. Note, however that deterministic replay alone is not sufficient for minimization: deterministic replay does not inform how the schedule space should be explored; it only allows one to deterministically replay prefixes of events. Moreover, minimizing a single deterministic replay log (without exploring divergent schedules) yields executions that are orders of magnitude larger than those produced by DEMi, as we discuss in §6.

Support for production traces. DEMi does not currently support minimization of production executions. DEMi requires that execution recordings are complete (meaning all message deliveries and external events are recorded) and partially ordered. Our current implementation achieves these properties simply by testing and minimizing on a single physical machine.

To support recordings from production executions, it should be possible to capture partial orders without requiring logical clocks on all messages: because the actor model only allows actors to process a single message at a time, we can compute a partial order simply by reconstructing message lineage from per-actor event logs (which record the order of messages received and sent by each actor). Crash-stop failures do not need to be recorded, since from the perspective of other processes these are equivalent to network partitions. Crash-recovery failures would need to be recorded to disk. Byzantine failures are outside the scope of our work.

Recording a sufficiently detailed log for each actor adds some logging overhead, but this overhead could be modest. For the systems we examined, Akka is primarily used as a control-plane, *not* a data-plane (e.g. Spark does not send bulk data over Akka), where recording overhead is not especially problematic.

4.2 Generality

We distinguish between the generality of the DEMi artifact, and the generality of our scheduling strategies.

Generality of DEMi. We targeted the Akka actor framework for one reason: thanks to the actor API (and to a lesser extent, AspectJ), we did not need to exert much engineering effort to interpose on (i) communication between processes, (ii) blocking operations, (iii) clocks, and (iv) remaining sources of non-determinism.

We believe that with enough interposition, it should be possible to sufficiently control other systems, regardless of language or programming model. That said, the effort needed to interpose could certainly be significant.

One way to increase the generality of DEMi would be to interpose at a lower layer (e.g. the network or syscall layer) rather than the application layer. This has several limitations. First, some of our scheduling strategies depend on application semantics (e.g. message types) which would be difficult to access at a lower layer. Transport layer complexities would also increase the size of the schedule space. Lastly, some amount of application layer interposition would still be necessary, e.g. interposition on user-level threads or blocking operations.

Generality of scheduling strategies. At their core, distributed systems are just concurrent systems (with the additional complexities of partial failure and asynchrony). Regardless of whether they are designed for multi-core or a distributed setting, the key property we assume from the program under test is that small schedules that are similar to original schedule should be likely to trigger the same invariant violation. To be sure, one can always construct adversarial counterexamples. Yet our results for two very different types of systems leave us optimistic that these scheduling strategies are broadly applicable.

5 Evaluation

Our evaluation focuses on two key metrics: (i) the size of the reproducing sequence found by DEMi, and (ii) how quickly DEMi is able to make minimization progress within a fixed time budget. We show a high-level overview of our results in Table 1. The "Bug Type" column shows two pieces of information: whether the bug can be triggered using TCP semantics (denoted as "FIFO") or whether it can only be triggered when UDP is used; and whether we discovered the bug ourselves or whether we reproduced a known bug. The "Provenance"

Bug Name	Bug Type	Initial	Provenance	STSSched	TFB	Optimal	NoDiverge
raft-45	Akka-FIFO, reproduced	2160 (E:108)	2138 (E:108)	1183 (E:8)	23 (E:8)	22 (E:8)	1826 (E:11)
raft-46	Akka-FIFO, reproduced	1250 (E:108)	1243 (E:108)	674 (E:8)	35 (E:8)	23 (E:6)	896 (E:9)
raft-56	Akka-FIFO, found	2380 (E:108)	2376 (E:108)	1427 (E:8)	82 (E:8)	21 (E:8)	2064 (E:9)
raft-58a	Akka-FIFO, found	2850 (E:108)	2824 (E:108)	953 (E:32)	226 (E:31)	51 (E:11)	2368 (E:35)
raft-58b	Akka-FIFO, found	1500 (E:208)	1496 (E:208)	164 (E:13)	40 (E:8)	28 (E:8)	1103 (E:13)
raft-42	Akka-FIFO, reproduced	1710 (E:208)	1695 (E:208)	1093 (E:39)	180 (E:21)	39 (E:16)	1264 (E:43)
raft-66	Akka-UDP, found	400 (E:68)	392 (E:68)	262 (E:23)	77 (E:15)	29 (E:10)	279 (E:23)
spark-2294	Akka-FIFO, reproduced	1000 (E:30)	886 (E:30)	43 (E:3)	40 (E:3)	25 (E:1)	43 (E:3)
spark-3150	Akka-FIFO, reproduced	600 (E:20)	536 (E:20)	18 (E:3)	14 (E:3)	11 (E:3)	18 (E:3)
spark-9256	Akka-FIFO, found (rare)	300 (E:20)	256 (E:20)	11 (E:1)	11 (E:1)	11 (E:1)	11 (E:1)

Table 1: Overview of case studies. "E:" is short for "Externals:". The 'Provenance', 'STSSched', and 'TFB' techniques are pipelined one after another. 'Initial' minus 'TFB' shows overall reduction; 'Provenance' shows how many events can be statically removed; 'STSSched' minus 'TFB' shows how our new techniques compare to the previous state of the art [70]; 'TFB' minus 'Optimal' shows how far from optimal our results are; and 'NoDiverge' shows the size of minimized executions when no divergent schedules are explored (explained in §6).

Bug Name	STSSched	TFB
raft-45	56s (594)	114s (2854)
raft-46	73s (384)	209s (4518)
raft-56	54s (524)	2078s (31149)
raft-58a	137s (624)	43345s (834972)
raft-58b	23s (340)	31s (1747)
raft-42	118s (568)	10558s (176517)
raft-66	14s (192)	334s (10334)
spark-2294	330s (248)	97s (78)
spark-3150	219s (174)	26s (21)
spark-9256	96s (73)	0s (0)

Table 2: Minimization runtime in seconds (total schedules executed). Overall runtime is the summation of "STSSched" and "TFB". spark-9256 only had unignorable events remaining after STSSched completed, so TFB was not necessary.

column shows how many events from the initial execution remain after statically pruning events that are concurrent with the safety violation. The "STSSched" column shows how many events remain after checking the initial schedules prescribed by our prior work [70] for each of delta debugging's subsequences. The "TFB" column shows the final execution size after we apply our techniques ('Type Fingerprints with Backtracks'), where we direct DPOR to explore as many backtrack points that match the types of original messages (but no other backtrack points) as possible within the 12 hour time budget we provided. Finally, the "Optimal" column shows the size of the smallest violation-producing execution we could construct by hand. We ran all experiments on a 2.8GHz Westmere processor with 16GB memory.

Overall we find that DEMi produces executions that are within a factor of 1X to 4.6X (1.6X median) the size of the smallest possible execution that triggers that bug, and between 1X and 16X (4X median) smaller than the executions produced by our previous technique (STSSched). STSSched is effective at minimizing external events (our primary minimization target) for most case studies. TFB is significantly more effective for minimizing internal events (our secondary target), especially for akka-raft. Replayable executions for all case studies are available at github.com/NetSys/demi-experiments.

We create the initial executions for all of our case studies by generating fuzz tests with DEMi (injecting a fixed number of random external events, and selecting internal messages to deliver in a random order) and selecting the first execution that triggers the invariant violation with $\geq$300 initial message deliveries. Fuzz testing terminated after finding a faulty execution within 10s of minutes for most of our case studies.

For case studies where the bug was previously known, we set up the initial test conditions (cluster configuration, external events) to closely match those described in the bug report. For cases where we discovered new bugs, we set up the test environment to explore situations that developers would likely encounter in production systems.

As noted in the introduction, the systems we focus on are akka-raft [3] and Apache Spark [90]. akka-raft, as an early-stage software project, demonstrates how DEMi can aid the development process. Our evaluation of Spark demonstrates that DEMi can be applied to complex, large scale distributed systems.

Reproducing Sequence Size. We compare the size of the minimized executions produced by DEMi against the smallest fault-inducing executions we could construct by hand (interactively instructing DEMi which messages to deliver). For 6 of our 10 case studies, DEMi was within a factor of 2 of optimal. There is still room for improvement however. For raft-58a for example, DEMi exhausted its time budget and produced an execution that was a factor of 4.6 from optimal. It could have found a smaller execution without exceeding its time budget with a better schedule exploration strategy.

Minimization Pace. To measure how quickly DEMi makes progress, we graph schedule size as a function of the number of executions DEMi tries. Figure 2 shows an example for raft-58b. The other case studies follow the same general pattern of sharply decreasing marginal gains.

We also show how much time (# of replays) DEMi took to reach completion of STSSched and TFB in Table 2.[7] The time budget we allotted to DEMi for all

[7] It is important to understand that DEMi is able to replay executions significantly more quickly than the original execution may have taken. This is because DEMi can trigger timer events before the wall-clock

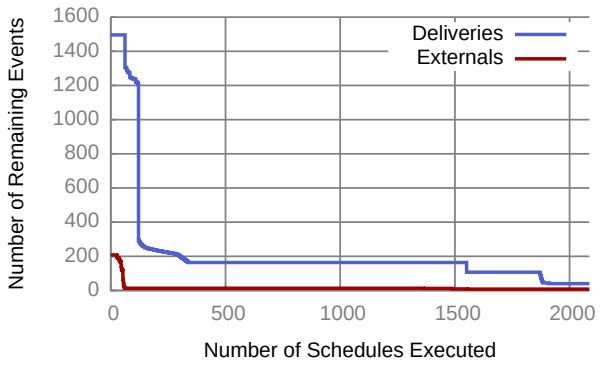

Figure 2: Minimization pace for raft-58b. Significant progress is made early on, then progress becomes rare.

case studies was 12 hours (43200s). All case studies except raft-56, raft-58a, and raft-42 reached completion of TFB in less than 10 minutes.

Qualitative Metrics. We do not evaluate how minimization helps with programmer productivity. Data on how humans do debugging is scarce; we are aware of only one study that measures how quickly developers debug minimized vs. non-minimized traces [40]. Nonetheless, since humans can only keep a small number of facts in working memory [62], minimization seems generally useful. As one developer puts it, "Automatically shrinking test cases to the minimal case is immensely helpful" [13].

5.1 Raft Case Studies

Our first set of case studies are taken from akka-raft [3]. akka-raft is implemented in 2,300 lines of Scala excluding tests. akka-raft has existing unit and integration tests, but it has not been deployed in production. The known bugs we reproduced had not yet been fixed; these were found by a recent manual audit of the code.

For full descriptions of each case study, see Appendix D. The lessons we took away from our akka-raft case studies are twofold. First, fuzz testing is quite effective for finding bugs in early-stage software. We found and fixed these bugs in less than two weeks, and several of the bugs would have been difficult to anticipate a priori. Second, debugging unminimized faulty executions would be very time consuming and conceptually challenging; we found that the most fruitful debugging process was to walk through events one-by-one to understand how the system arrived at the unsafe state, which would take hours for unminimized executions.

5.2 Spark Case Studies

Spark [4] is a mature software project, used widely in production. The version of Spark we used for our evaluation consists of more than 30,000 lines of Scala for just the core execution engine. Spark is also interesting because it has a significantly different communication pattern than Raft (e.g., statically defined masters).

For a description of our Spark case studies, see Appendix E. Our main takeaway from Spark is that for the simple Spark jobs we submitted, STSSched does surprisingly well. We believe this is because Spark's communication tasks were all almost entirely independent of each other. If we had submitted more complex Spark jobs with more dependencies between messages (e.g. jobs that make use of intermediate caching between stages) STSSched likely would not have performed as well.

duration for those timers has actually passed, without the application being aware of this fact (cf. [39])

	Without Shrinking	With shrinking
Initial Events	360 (E: 9 bootstraps)	360 (E: 9 bootstraps)
After STSSched	81 (E: 8 bootstraps)	51 (E: 5 bootstraps)

Table 3: External message shrinking results for raft-45 starting with 9 processes. Message shrinking + minimization was able to reduce the cluster size to 5 processes.

	akka-raft	Spark
Message Fingerprint	59	56
Non-Determinism	2	~250
Invariants	331	151
Test Configuration	328	445

Table 4: Complexity (lines of Scala code) needed to define message fingerprints, mitigate non-determinism, define invariants, and configure DEMi. Akka API interposition (336 lines of AspectJ) is application independent.

5.3 Auxiliary Evaluation

External message shrinking. We demonstrate the benefits of external message shrinking with an akka-raft case study. Recall that akka-raft processes receive an external bootstrapping message that informs them of the IDs of all other processes. We started with a 9 node akka-raft cluster, where we triggered the raft-45 bug. We then shrank message contents by removing each element (process ID) of bootstrap messages, replaying these along with all other events in the failing execution, and checking whether the violation was still triggered. We were able to shrink the bootstrap message contents from 9 process IDs to 5 process IDs. Finally, we ran STSSched to completion, and compared the output to STSSched without the initial message shrinking. The results shown in Table 3 demonstrate that message shrinking can help minimize both external events and message contents.

Instrumentation Overhead. Table 4 shows the complexity in terms of lines of Scala code needed to define message fingerprint functions, mitigate non-determinism (with the application modifications described in §4), specify invariants, and configure DEMi. In total we spent roughly one person-month debugging non-determinism.

6 Related Work

We start this section with a discussion of the most closely related literature. We focus only on DEMi's minimization techniques, since DEMi's interposition and testing functionality is similar to other systems [55, 57, 72].

Input Minimization for Sequential Programs. Mini-

mization algorithms for sequentially processed inputs are well-studied [18,20,24,40,69,81,94]. These form a component of our solution, but they do not consider interleavings of internal events from concurrent processes.

Minimization without Interposition. Several tools minimize inputs to concurrent systems without controlling sources of non-determinism [10, 26, 44, 47, 76]. The most sophisticated of these replay each subsequence multiple times and check whether the violation is reproduced at least once [25,44]. Their major advantage is that they avoid the engineering effort required to interpose. However, we found in previous work [70] that bugs are often not easily reproducible without interposition.

QuickCheck's PULSE controls the message delivery schedule [25] and supports schedule minimization. During replay, it only considers the order messages are sent in, not message contents. When it cannot replay a step, it skips it (similar to STSSched), and reverts to random scheduling once expected messages are exhausted [43].

Thread Schedule Minimization. Other techniques seek to minimize thread interleavings leading up to concurrency bugs [22, 30, 41, 46]. These generally work by iteratively feeding a single input (the thread schedule) to a single entity (a deterministic scheduler). These approaches ensure that they never diverge from the original schedule (otherwise the recorded context switch points from the original execution would become useless). Besides minimizing context switches, these approaches at best *truncate* thread executions by having threads exit earlier than they did in the original execution.

Program Analysis. By analyzing the program's control- and dataflow dependencies, one can remove events in the middle of the deterministic replay log without causing divergence [19, 31, 42, 54, 74, 79]. These techniques do not explore alternate code paths. Program analysis also over-approximates reachability, disallowing them from removing dependencies that actually commute.

We compare against these techniques by configuring DEMi to minimize as before, but abort any execution where it detects a previously unobserved state transition. Column 'NoDiverge' of Table 1 shows the results, which demonstrate that divergent executions are crucial to DEMi's reduction gains for the akka-raft case studies.

Model Checking. Algorithmically, our work is most closely related to the model checking literature.

Abstract model checkers convert (concurrent) programs to logical formulas, find logical contradictions (invariant violations) using solvers, and minimize the logical conjunctions to aid understanding [23,49,61]. Model checkers are very powerful, but they are typically tied to a single language, and assume access to source code, whereas the systems we target (e.g. Spark) are composed of multiple languages and may use proprietary libraries.

It is also possible to extract formulas from raw binaries [11]. Fuzz testing is significantly lighter weight.

If, rather than randomly fuzzing, testers enumerated inputs of progressively larger sizes, failing tests would be minimal by construction. However, breadth first enumeration takes very long to get to 'interesting' inputs (After 24 hours of execution, our bounded DPOR implementation with depth bound slightly greater than the optimal trace size still had not found any invariant violations. In contrast, DEMi's randomized testing discovered most of our reported bugs within 10s of minutes). Furthermore, minimization is useful beyond testing, e.g. for simplifying production traces.

Because systematic input enumeration is intractable, many papers develop heuristics for finding bugs quickly [17, 28, 35, 55, 57, 63, 64, 66, 75, 78, 84]. We do the same, but crucially, we are able to use information from previously failing executions to guide our search.

As far as we know, we are the first to combine DPOR and delta debugging to minimize executions. Others have modified DPOR to keep state [87, 88] and to apply heuristics for choosing initial schedules [53], but these changes are intended to help find new bugs.

Bug Reproduction. Several papers seek to find a schedule that reproduces a given concurrency bug [9, 67, 91, 92]. These do not seek to find a minimal schedule.

Probabilistic Diagnosis. To avoid the runtime overhead of deterministic replay, other techniques capture carefully selected diagnostic information from production execution(s), and correlate this information to provide best guesses at the root causes of bugs [12,21,48,68,89]. We assume more complete runtime instrumentation (during testing), but provide exact reproducing scenarios.

Log Comprehension. Model inference techniques summarize log files in order to make them more easily understandable by humans [14–16,32,58,59]. Model inference is complementary, as it does not modify the event logs.

Program Slicing & Automated Debugging. Program slicing [80] and the subsequent literature on automated debugging [27, 45, 60, 73, 83, 95] seek to localize errors in the code itself. Our goal is to slice the temporal dimension of an execution rather than the code dimension.

7 Conclusion

Distributed systems, like most software systems, are becoming increasingly complex over time. In comparison to other areas of software engineering however, the development tools that help programmers cope with the complexity of distributed & concurrent systems are lagging behind their sequential counterparts. Inspired by the obvious utility of test case reduction tools, we sought to develop a minimization tool for distributed executions. Our evaluation results for two very different systems leave us optimistic that these techniques can be successfully applied to a wide range of concurrent systems.

References

[1] 7 Tips for Fuzzing Firefox More Effectively. https://blog.mozilla.org/security/2012/06/20/7-tips-for-fuzzing-firefox-more-effectively/.

[2] Akka official website. http://akka.io/.

[3] akka-raft Github repo. https://github.com/ktoso/akka-raft.

[4] Apache Spark Github repo. https://github.com/apache/spark/.

[5] DEMi Github repo. https://github.com/NetSys/demi.

[6] GNU's guide to testcase reduction. https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction.

[7] LLVM bugpoint tool: design and usage. http://llvm.org/docs/Bugpoint.html.

[8] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. International Workshop on Distributed Algorithms '97.

[9] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. SOSP '09.

[10] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq QuickCheck. Erlang '06.

[11] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. ICSE '14.

[12] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. OSDI '04.

[13] Basho Blog. QuickChecking Poolboy for Fun and Profit. http://tinyurl.com/qgc387k.

[14] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring Models of Concurrent Systems from Logs of their Behavior with CSight. ICSE '14.

[15] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. ESEC/FSE '11.

[16] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of their Behavior. IEEE ToC '72.

[17] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. ASPLOS '10.

[18] M. Burger and A. Zeller. Minimizing Reproduction of Software Failures. ISSTA '11.

[19] Y. Cai and W. Chan. Lock Trace Reduction for Multithreaded Programs. TPDS '13.

[20] K.-h. Chang, V. Bertacco, and I. L. Markov. Simulation-Based Bug Trace Minimization with BMC-Based Refinement. IEEE TCAD '07.

[21] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, O. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. DSN '02.

[22] J. Choi and A. Zeller. Isolating Failure-Inducing Thread Schedules. SIGSOFT '02.

[23] J. Christ, E. Ermis, M. Schäf, and T. Wies. Flow-Sensitive Fault Localization. VMCAI '13.

[24] K. Claessen and J. Hughes. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. ICFP '00.

[25] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding Race Conditions in Erlang with QuickCheck and PULSE. ICFP '09.

[26] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. ICSE '07.

[27] H. Cleve and A. Zeller. Locating Causes of Program Failures. ICSE '05.

[28] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: Effective Unit Testing for Concurrency Libraries. PPoPP '10.

[29] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. OSDI '02.

[30] M. A. El-Zawawy and M. N. Alanazi. An Efficient Binary Technique for Frace Simplifications of Concurrent Programs. ICAST '14.

[31] A. Elyasov, I. W. B. Prasetya, and J. Hage. Guided Algebraic Specification Mining for Failure Simplification. TSS '13.

[32] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. IEEE ToSE '01.

[33] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. JACM '85.

[34] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. POPL '05.

[35] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. OSDI '14.

[36] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging For Distributed Applications. ATC '06.

[37] P. Godefroid and N. Nagappan. Concurrency at Microsoft - An Exploratory Survey. CAV '08.

[38] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. PhD Thesis, '95.

[39] D. Gupta, K. Yocum, M. Mcnett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. NSDI '06.

[40] M. Hammoudi, B. Burg, Gigon, and G. Rothermel. On the Use of Delta Debugging to Reduce Recordings and Facilitate Debugging of Web Applications. ESEC/FSE '15.

[41] J. Huang and C. Zhang. An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs. SAS '11.

[42] J. Huang and C. Zhang. LEAN: Simplifying Concurrency Bug Reproduction via Replay-Supported Execution Reduction. OOPSLA '12.

[43] J. M. Hughes. Personal Communication.

[44] J. M. Hughes and H. Bolinder. Testing a Database for Race Conditions with QuickCheck. Erlang '11.

[45] J. A. Jones and M. J. Harrold and J. Stasko. Visualization of Test Information To Assist Fault Localization. ICSE '02.

[46] N. Jalbert and K. Sen. A Trace Simplification Technique for Effective Debugging of Concurrent Programs. FSE '10.

[47] W. Jin and A. Orso. F3: Fault Localization for Field Failures. ISSTA '13.

[48] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures. SOSP '15.

[49] S. Khoshnood, M. Kusano, and C. Wang. ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrency Bugs. ISSTA '15.

[50] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. ECOOP '01.

[51] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. CACM '78.

[52] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: a Study of Developer Work Habits. ICSE '06.

[53] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. FASE '10.

[54] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang. Toward Generating Reducible Replay Logs. PLDI '11.

[55] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. OSDI '14.

[56] C.-C. Lin, V. Jalaparti, M. Caesar, and J. Van der Merwe. DEFINED: Deterministic Execution for Interactive Control-Plane Debugging. ATC '13.

[57] H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. NSDI '09.

[58] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. ICSE '08.

[59] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining Invariants from Console Logs for System Problem Detection. ATC '10.

[60] M. Jose and R. Majmudar. Cause Clue Causes: Error Localization Using Maximum Satisfiability. PLDI '11.

[61] N. Machado, B. Lucia, and L. Rodrigues. Concurrency Debugging with Differential Schedule Projections. PLDI '15.

[62] G. A. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psychological Review '56.

[63] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. PLDI '07.

[64] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. SOSP '08.

[65] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. ATC '14.

[66] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from their Hiding Places. ASPLOS '09.

[67] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. SOSP '09.

[68] S. M. Park. *Effective Fault Localization Techniques for Concurrent Software*. PhD Thesis, '14.

[69] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case Reduction for C Compiler Bugs. PLDI '12.

[70] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. SIGCOMM '14.

[71] O. Shacham, E. Yahav, G. G. Gueta, A. Aiken, N. Bronson, M. Sagiv, and M. Vechev. Verifying Atomicity via Data Independence. ISSTA'14.

[72] J. Simsa, R. Bryant, and G. A. Gibson. dBug: Systematic Evaluation of Distributed Systems. SSV '10.

[73] W. Sumner and X. Zhang. Comparative Causality: Explaining the Differences Between Executions. ICSE '13.

[74] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling Tracing of Long-Running Multithreaded Programs via Dynamic Execution Reduction. ISSTA '07.

[75] V. Terragni, S.-C. Cheung, and C. Zhang. RECONTEST: Effective Regression Testing of Concurrent Programs. ICSE '15.

[76] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing Production Run Failures at the User's Site. SOSP '07.

[77] Twitter Blog. Diffy: Testing Services Without Writing Tests. https://blog.twitter.com/2015/diffy-testing-services-without-writing-tests.

[78] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting Where it Hurts: An Automatic Concurrent Debugging Technique. ISSTA '07.

[79] J. Wang, W. Dou, C. Gao, and J. Wei. Fast Reproducing Web Application Errors. ISSRE '15.

[80] M. Weiser. Program Slicing. ICSE '81.

[81] A. Whitaker, R. Cox, and S. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. SOSP '04.

[82] P. Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. POPL '86.

[83] J. Xuan and M. Monperrus. Test Case Purification for Improving Fault Localization. FSE '14.

[84] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. NSDI '09.

[85] M. Yabandeh and D. Kostic. DPOR-DS: Dynamic Partial Order Reduction in Distributed Systems. 2009 Tech Report.

[86] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. PLDI '11.

[87] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. MCS '08.

[88] X. Yi, J. Wang, and X. Yang. Stateful Dynamic Partial-Order Reduction. FMSE '06.

[89] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. ASPLOS '10.

[90] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI '12.

[91] C. Zamfir, G. Altekar, G. Candea, and I. Stoica. Debug Determinism: The Sweet Spot for Replay-Based Debugging. HotOS '11.

[92] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. EuroSys '10.

[93] A. Zeller. Yesterday, my program worked. Today, it does not. Why? ESEC/FSE '99.

[94] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE TSE '02.

[95] S. Zhang and C. Zhang. Software Bug Localization with Markov Logic. ICSE '14.

A Delta Debugging

We show the Delta Debugging simplification algorithm [93] we use in Figure 3, and an example execution of Delta Debugging in Figure 5. An updated version of the ddmin simplification algorithm appeared in [94]. We use the simpler version of ddmin (which is equivalent to the version ddmin from [94], except that it does not consider complements) because we ensure that each subsequence of external events is consistent (semantically valid), and therefore are still guarenteed to find a 1-minimal output without needing to consider complements.

B DPOR

We show the original depth-first version of Dynamic Partial Order Reduction in Algorithm 2. Our modified DPOR algorithm uses a priority queue rather than a (recursive) stack, and tracks which schedules it has explored in the past. Tracking which schedules we have explored in the past is necessary to avoid exploring redundant schedules (an artifact of our non depth-first exploration order). The memory footprint required for tracking previously explored schedules continues growing for every new schedule we explore. Because we assume a fixed time budget though, we typically exhaust our time budget before DEMi runs out of memory.

There are a few desirable properties of DPOR we want to maintain, despite our prioritized exploration order:
Soundness: any executed schedule should be valid, i.e. possible to execute on an uninstrumented version of the program starting from the initial configuration.

Step	External Event Subsequence								TEST
1	e_1	e_2	e_3	e_4	·	·	·	·	✔
2	·	·	·	·	e_5	e_6	e_7	e_8	✔
3	e_1	e_2	·	·	e_5	e_6	e_7	e_8	✔
4	·	·	e_3	e_4	e_5	e_6	e_7	e_8	✘
5	·	·	e_3	·	e_5	e_6	e_7	e_8	✘ (e_3 found)
6	e_1	e_2	e_3	e_4	e_5	e_6	·	·	✘
7	e_1	e_2	e_3	e_4	e_5	·	·	·	✔ (e_6 found)
Result	·	·	e_3	·	·	e_6	·	·	

Table 5: Example execution of Delta Debugging, taken from [93]. '·' denotes an excluded original external event.

Programmer-provided Specification	Default
Initial cluster configuration	-
External event probabilities	No external events
Invariants	Uncaught exceptions
Violation fingerprint	Match on any violation
Message fingerprint function	Match on message type
Non-determinism mitigation	Replay multiple times

Table 6: Tasks we assume the application programmer completes in order to test and minimize using DEMi. Defaults of '-' imply that the task is not optional.

Efficiency: the happens-before partial order for every executed schedule should never be a prefix of any other partial orders that have been previously explored.
Completeness: when the state space is acyclic, the strategy is guaranteed to find every possible safety violation.

Because we experimentally execute each schedule, soundness is easy to ensure (we simply ensure that we do not violate TCP semantics if the application assumes TCP, and we make sure that we cancel timers whenever the application asks to do so). Improved efficiency is the main contribution of partial order reduction. The last property—completeness—holds for our modified version of DPOR so long as we always set at least as many backtrack points as depth-first DPOR.

C Programmer Effort

In Table 6 we summarize the various tasks, both optional and necessary, that we assume programmers complete in order to test and minimize using DEMi.

D Raft Case Studies

Raft is a consensus protocol, designed to replicate a fault tolerant linearizable log of client operations. akka-raft is an open source implementation of Raft.

The external events we inject for akka-raft case studies are bootstrap messages (which processes use for discovery of cluster members) and client transaction requests. Crash-stop failures are indirectly triggered through fuzz schedules that emulate network partitions. The cluster size was 4 nodes (quorum size=3) for all akka-raft case studies.

The invariants we checked for akka-raft are the consensus invariants specified in Figure 3 of the Raft paper [65]: Election Safety (at most one leader can be elected in a given term), Log Matching (if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index), Leader Completeness (if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms), and State Machine Safety (if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index). Note that a violation of any of these invariants allows for the possibility for the system to later violate the main linearizability in-

Input: E s.t. E is a sequence of externals, and $\text{TEST}(E) = \text{✘}$. Output: $E' = ddmin(E)$ s.t. $E' \sqsubseteq E$, $\text{TEST}(E') = \text{✘}$, and E' is minimal.

$$ddmin(E) = ddmin_2(E, \emptyset) \quad \text{where}$$

$$ddmin_2(E', R) = \begin{cases} E' & \text{if } |E'| = 1 \text{ (“base case”)} \\ ddmin_2(E_1, R) & \text{else if } \text{TEST}(E_1 \cup R) = \text{✘} \text{ (“in } E_1\text{”)} \\ ddmin_2(E_2, R) & \text{else if } \text{TEST}(E_2 \cup R) = \text{✘} \text{ (“in } E_2\text{”)} \\ ddmin_2(E_1, E_2 \cup R) \cup ddmin_2(E_2, E_1 \cup R) & \text{otherwise (“interference”)} \end{cases}$$

where ✘ denotes an invariant violation, $E_1 \sqsubset E'$, $E_2 \sqsubset E'$, $E_1 \cup E_2 = E'$, $E_1 \cap E_2 = \emptyset$, and $|E_1| \approx |E_2| \approx |E'|/2$ hold.

Figure 3: Delta Debugging Algorithm from [93]. $\sqsubseteq$ and $\sqsubset$ denote subsequence relations. TEST is defined in Algorithm 1.

Algorithm 2 The original depth-first version of Dynamic Partial Order Reduction from [34]. $last(S)$ denotes the configuration reached after executing S; $next(\kappa, m)$ denotes the state transition (message delivery) where the message m is processed in configuration κ; $\rightarrow_S$ denotes ‘happens-before’; $pre(S, i)$ refers to the configuration where the transition t_i is executed; $dom(S)$ means the set $\{1, \ldots, n\}$; $S.t$ denotes S extended with an additional transition t.

Initially: EXPLORE($\emptyset$)
procedure EXPLORE(S)
  $\kappa \leftarrow last(S)$
  for each message $m \in pending(\kappa)$ **do**
    if $\exists i = max(\{i \in dom(S) | S_i \text{ is dependent and may be coenabled with } next(\kappa, m) \text{ and } i \not\rightarrow_S m\})$**:**
      $E \leftarrow \{m' \in enabled(pre(S, i)) | m' = m \text{ or } \exists j \in dom(S) : j > i \text{ and } m' = msg(S_j) \text{ and } j \rightarrow_S m\}$
      if $E \neq \emptyset$**:**
        add any $m' \in E$ to $backtrack(pre(S, i))$
      else
        add all $m \in enabled(pre(S, i))$ to $backtrack(pre(S, i))$
  if $\exists m \in enabled(\kappa)$**:**
    $backtrack(\kappa) \leftarrow \{m\}$
    $done \leftarrow \emptyset$
    while $\exists m \in (backtrack(\kappa) \setminus done)$ **do**
      add m to $done$
      EXPLORE($S.next(\kappa, m)$)

variant (State Machine Safety).

For each of the bugs where we did not initially know the root cause, we started debugging by first minimizing the failing execution. Then, we walked through the sequence of message deliveries in the minimized execution. At each step, we noted the current state of the actor receiving the message. Based on our knowledge of the way Raft is supposed to work, we found places in the execution that deviate from our understanding of correct behavior. We then examined the code to understand why it deviated, and came up with a fix. Finally, we replayed to verify the bug fix.

The akka-raft case studies in Table 1 are shown in the order that we found or reproduced them. To prevent bug causes from interfering with each other, we fixed all other known bugs for each case study. We reported all bugs and fixes to the akka-raft developers.

raft-45: Candidates accept duplicate votes from the same election term. Raft is specified as a state machine with three states: Follower, Candidate, and Leader. Candidates attempt to get themselves elected as leader by soliciting a quorum of votes from their peers in a given election term (epoch).

In one of our early fuzz runs, we found a violation of ‘Leader Safety’, i.e. two processes believed they were leader in the same election term. This is a highly problematic situation for Raft to be in, since the leaders may overwrite each others’ log entries, thereby violating the key linearizability guarantee that Raft is supposed to provide.

The root cause for this bug was that akka-raft’s candidate state did not detect duplicate votes from the same follower in the same election term. (A follower might resend votes because it believed that an earlier vote was dropped by the network). Upon receiving the duplicate vote, the candidate counts it as a new vote and steps up to leader before it actually achieved a quorum of votes.

raft-46: Processes neglect to ignore certain votes from previous terms. After fixing the previous bug, we found another execution where two leaders were elected in the same term.

In Raft, processes attach an ‘election term’ number to all messages they send. Receiving processes are supposed to ignore any messages that contain an election term that is lower than what they believe is the current term.

akka-raft properly ignored lagging term numbers for some, but not all message types. DEMi delayed the delivery of messages from previous terms and uncovered a case where a candidate incorrectly accepted a vote message from a previous election term.

raft-56: Nodes forget who they voted for. akka-raft is written as a finite state machine. When making a state transition, FSM processes specify both which state they want to transition to, and which instance variables they want to keep once they have transitioned.

All of the state transitions for akka-raft were correct except one: when the Candidate steps down to Follower (e.g., because it receives an 'AppendEntries' message, indicating that there is another leader in the cluster), it *forgets* which node it previously voted for in that term. Now, if another node requests a vote from it in the same term, it may vote for a different node than it previously voted for in the same term, later causing two leaders to be elected, i.e. a violation of Raft's "Leader Safety" condition. We discovered this by manually examining the state transitions made by each process throughout the minimized execution.

raft-58a: Pending client commands delivered before initialization occurs. After ironing out leader election issues, we started finding other issues. In one of our fuzz runs, we found that a leader process threw an assertion error.

When an akka-raft Candidate first makes the state transition to leader, it does not immediately initialize its state (the 'nextIndex' and 'matchIndex' variables). It instead sends a message to itself, and initializes its state when it receives that self-message.

Through fuzz testing, we found that it is possible that the Candidate could have pending ClientCommand messages in its mailbox, placed there before the Candidate transitioned to Leader and sent itself the initialization message. Once in the Leader state, the Akka runtime will first deliver the ClientCommand message. Upon processing the ClientCommand message the Leader tries to replicate it to the rest of the cluster, and updates its nextIndex hashmap. Next, when the Akka runtime delivers the initialization self-message, it will overwrite the value of nextIndex. When it reads from nextIndex later, it is possible for it to throw an assertion error because the nextIndex values are inconsistent with the contents of the Leader's log.

raft-58b: Ambiguous log indexing. In one of our fuzz tests, we found a case where the 'Log Matching' invariant was violated, i.e. log entries did not appear in the same order on all machines.

According to the Raft paper, followers should reject AppendEntries requests from leaders that are behind, i.e. prevLogIndex and prevLogTerm for the AppendEntries message are behind what the follower has in its log. The leader should continue decrementing its nextIndex hashmap until the followers stop rejecting its AppendEntries attempts.

This should have happened in akka-raft too, except for one hiccup: akka-raft decided to adopt 0-indexed logs, rather than 1-indexed logs as the paper suggests. This creates a problem: the initial value of prevLogIndex is ambiguous: Followers can not distinguish between an AppendEntries for an empty log (prevLogIndex == 0) an AppendEntries for the leader's 1st command (prevLogIndex == 0), and an AppendEntries for the leader's 2nd command (prevLogIndex == 1 1 == 0). The last two cases need to be distinguishable. Otherwise followers will not be able to reject inconsistent logs. This corner would have been hard to anticipate; at first glance it seems fine to adopt the convention that logs should be 0-indexed instead of 1-indexed.

As a result of this ambiguity, followers were unable to correctly reject AppendEntries requests from leader that were behind.

raft-42: Quorum computed incorrectly. We also found a fuzz test that ended in a violation of the 'Leader Completeness' invariant, i.e. a newly elected leader had a log that was irrecoverably inconsistent with the logs of previous leaders.

Leaders are supposed to commit log entries to their state machine when they knows that a quorum (N/2+1) of the processes in the cluster have that entry replicated in their logs. akka-raft had a bug where it computed the highest replicated log index incorrectly. First it sorted the values of matchIndex (which denote the highest log entry index known to be replicated on each peer). But rather than computing the median (or more specifically, the N/2+1'st) of the sorted entries, it computed the mode of the sorted entries. This caused the leader to commit entries too early, before a quorum actually had that entry replicated. In our fuzz test, message delays allowed another leader to become elected, but it did not have all committed entries in its log due to the previously leader committing too soon.

As we walked through the minimized execution, it became clear mid-way through the execution that not all entries were fully replicated when the master committed its first entry. Another process without all replicated entries then became leader, which constituted a violation of the "Leader Completeness" invariant.

raft-66: Followers unnecessarily overwrite log entries. The last issue we found is only possible to trigger if the underlying transport protocol is UDP, since it requires reorderings of messages between the same source, destination pair. The akka-raft developers say they do not currently support UDP, but they would like to adopt UDP in the future due to its lower latency.

The invariant violation here was a violation of the

'Leader Completeness' safety property, where a leader is elected that does not have all of the needed log entries.

Leaders replicate uncommitted ClientCommands to the rest of the cluster in batches. Suppose a follower with an empty log receives an AppendEntries containing two entries. The follower appends these to its log.

Then the follower subsequently receives an AppendEntries containing only the first of the previous two entries (this message was delayed). The follower will inadvertently delete the second entry from its log.

This is not just a performance issue: after receiving an ACK from the follower, the leader is under the impression that the follower has two entries in its log. The leader may have decided to commit both entries if a quorum was achieved. If another leader becomes elected, it will not necessarily have both committed entries in its log as it should, leading to a 'LeaderCompleteness' violation.

E Spark Case Studies

Spark is a large scale data analytics framework. We focused our efforts on reproducing known bugs in the core Spark engine, which is responsible for orchestrating computation across multiple machines.

We looked at the entire history of bugs reported for Spark's core engine. We found that most reported bugs only involve sequential computation on a single machine (e.g. crashes due to unexpected user input). We instead focused on reported bugs involving concurrency across machines or partial failures. Of the several dozen reported concurrency or partial failure bugs, we chose three.

The external events we inject for Spark case studies are worker join events (where worker nodes join the cluster and register themselves with the master), job submissions, and crash-recoveries of the master node. The Spark job we ran for all case studies was a simple parallel approximation of the digits of Pi.

spark-2294: Locality inversion. In Spark, an 'executor' is responsible for performing computation for Spark jobs. Spark jobs are assigned 'locality' preferences: the Spark scheduler is supposed to launch 'NODE_LOCAL' tasks (where the input data for the task is located on the same machine) before launching tasks without preferences. Tasks without locality preferences are in turn supposed to be launched before 'speculative' tasks.

The bug for this case study was the following: if an executor E is free, a task may be speculatively assigned to E when there are other tasks in the job that have not been launched (at all) yet. Similarly, a task without any locality preferences may be assigned to E when there was another 'NODE_LOCAL' task that could have been scheduled. The root cause of this bug was an error in Spark scheduler's logic: under certain configurations of pending Spark jobs and currently available executors, the Spark scheduler would incorrectly invert the locality priorities. We reproduced this bug by injecting random, concurrently running Spark jobs (with differing locality preferences) and random worker join events.

spark-3150: Simultaneous failure causes infinite restart loop. Spark's master node supports a 'ColdReplication' mode, where it commits its state to a database (e.g., ZooKeeper). Whenever the master node crashes, the node that replaces it can read that information from the database to bootstrap its knowledge of the cluster state.

To trigger this bug, the master node and the driver process need to fail simultaneously. When the master node restarts, it tries to read its state from the database. When the driver crashes simultaneously, the information the master reads from the database is corrupted: some of the pointers referencing information about the driver are null. When the master reads this information, it dereferences a null pointer and crashes again. After failing, the master restarts, tries to recover its state, and crashes in an infinite cycle. The minimized execution for this bug contained exactly these 3 external events, which made the problematic code path immediately apparent.

spark-9256: Delayed message causes master crash. We found the following bug through fuzz testing.

As part of initialization, Spark's client driver registers with the Master node by repeatedly sending a RegisterApplication message until it receives a RegisteredApplication response. If the RegisteredApplication response is delayed by at least as long as the configured timeout value (or if the network duplicates the RegisterApplication RPC), it is possible for the Master to receive two RegisterApplication messages for the same client driver.

Upon receiving the second RegisterApplication message, the master attempts to persist information about the client driver to disk. Since the file containing information about the client driver already exists though, the master crashes with an IllegalStateException.

This bug is possible to trigger in production, but it will occur only very rarely. The name of the file containing information has a second-granularity timestamp associated with it, so it would only be possible to have a duplicate file if the second RegisteredApplication response arrived in the same second as the first response.

Acknowledgements

We thank our shepherd David Lie and the anonymous reviewers for their feedback. We also thank Peter Alvaro, Barath Raghavan, and Kay Ousterhout for feedback on the submitted draft. This research was supported by NSF CNS 1040838 and a gift from Intel. Colin Scott was also supported by an NSF Graduate Research Fellowship.