Minimizing Faulty Executions of Distributed Systems

Colin Scott Aurojit Panda Vjekoslav Brajkovic George Necula
Arvind Krishnamurthy[†] Scott Shenker
UC Berkeley [†]*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). In this paper 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 achieves up to 97% reduction of execution length within a small time budget.

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, 82, 90] have already proven themselves highly valuable, and are routinely applied to bug reports for software projects such as Firefox [1], LLVM [8], and GCC [7].

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., node failures) are *interleaved* with the internal events (e.g. intra-process message deliveries) of concurrent processes. Minimization algorithms must therefore consider both which of the 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 [68], 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 [68]. Here we target a broader range of systems, formally define our problem statement, and exercise significantly greater control over execution in order to 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 implementation, and Apache Spark [86], a widely used data analytics framework. Across 11 known and discovered bugs, DEMi reduces the number of events in faulty execution traces by as much as 97%, and provides results that are up to 16 times smaller than those produced by the state-

[1] Developers spend a significant portion of their time debugging (49% of their time according to one study [50]), especially when the bugs involve concurrency (70% of reported concurrency bugs in [35] took days to months to fix).

of-the-art blackbox minimization technique [68]. The results we find for these two very different distributed systems leave us optimistic that these techniques, along with adequate visibility into events (either through a framework such as 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, we believe this model is general enough to capture the behavior of many practical systems.

2.1 System Model

Following [31], 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;[2] 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* explicitly consider fail-stop process 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] though it may maintain non-volatile state.

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 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 (if any), and the interleavings of internal events that appeared during the execution.

After generating a faulty execution, we minimize it by executing modified versions of the original fault-inducing schedule and checking whether they still trigger the invariant violation. We must always ensure that each modified schedule we execute is valid, meaning that whenever an internal message is to be delivered, that message is pending. We must additionally ensure that external events have their causal dependencies met, i.e. we ensure that forced restarts are always preceded by a start event for that process. We assume that external messages are independent of each other, i.e., we do not support external messages that, when removed, cause some other external event to become invalid.

2.3 Problem Definition

We are given a schedule τ injected by a test orchestrator,[3] 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 invariant 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 reproduces $\overline{P}$, but if we were to remove any single external event e from E', there would not exist any schedules containing $E' - e$ that reproduce $\overline{P}$.[4]

We start by minimizing external (input) events because they are the first level of abstraction that developers reason about. After finding an MCS, we turn to minimizing internal events, by searching for smaller schedules containing the same external events as the MCS that still triggers the invariant violation. Lastly, we seek to minimize the contents (e.g. data payloads) of external messages.

[3] We explain how we generate these schedules in §4.

[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 [90], we only require a 1-minimal subsequence E', an approximation of global minimality.

Note that we do not focus on bugs involving only sequential computation on a single process. Moreover, we do not specifically focus on human misconfigurations, but instead observe the behavior of the distributed system after it has received an initial configuration. Our focus is on minimizing concurrent schedules, where the complexity of distributed systems lies.

With a minimized execution in hand, the developer begins debugging. Echoing the benefits of sequential test 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 schedule of a fixed length containing the given external events. The globally minimal MCS would then be the shortest prefix 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 a reasonable time, we split schedule exploration into two parts. We start by using delta debugging [90], an input minimization algorithm similar to binary search, to prune extraneous external events. Delta debugging works by picking subsequences of the external events, and checking whether each subsequence still triggers the safety violation. We assume the user gives us a maximum time budget, and we spread this time budget evenly across each subsequence's exploration.

To check whether a particular subsequence of external events results in the specified safety violation, we need to explore the space of possible interleavings of internal events and external events. We use Dynamic Partial Order Reduction (DPOR) to prune this schedule space by eliminating equivalent schedules (i.e. schedules that differ only in the ordering of commutative events [32]). 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 schedules in 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. 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 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 distinct executions?*

We implement several *match* functions that tell us whether pending messages from the original execution correspond to the same logical message from the current execution. We then start our exploration with a single, uniquely-defined schedule that closely resembles the original faulty execution. If this schedule does not reproduce the violation, we begin exploring nearby schedules. We stop checking whether a particular external event subsequence triggers the violation 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 to be tested using 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 to check whether the resulting subsequence, E', still triggers the violation. If E' triggers the violation, then we can safely 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. However, this would require that we check $O(|E|)$ subsequences to determine

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

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

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

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 1: Delta Debugging Algorithm from [89]. $\sqsubseteq$ and $\sqsubset$ denote subsequence relations. TEST is defined in Algorithm 1.

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 [90], an algorithm similar to binary search, to achieve $O(log(|E|))$ average case runtime.[5] The delta debugging algorithm is shown in Figure 1.

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 [36], 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 [49]: 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) [32,36] 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) [32] is a refinement of POR: at each step, it picks a pending message to deliver, dynamically computes which of the 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 deliver the other pending messages at the same point of the execution 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 can easily get stuck exploring a small portion of the schedule space [55]. We address this problem in two ways: first, as mentioned before, we limit *ExtMin* so it spreads its fixed time budget 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.

3.3 Schedule Exploration Strategies

We guide schedule exploration by manipulating two degrees of freedom within DPOR: (i) we carefully choose the 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 tracking the schedules we have already explored, we can pick backtrack points in a prioritized order without exploring redundant schedules.

A scheduling strategy implements a particular backtrack prioritization order. Scheduling strategies return the first violation-reproducing schedule they can find, or null if they cannot find any reproducing schedules within a fixed time budget. We design our scheduling strategies 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 struc-

[5] Since we assume that external events are causally independent, we can use the version of delta debugging that avoids checking supersets (defined in [89]) and has worst case complexity of $O(|E|)$.

tures 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 pending (i.e. sent previously by some actor) at the point that internal message should be delivered, we skip it over 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 message. 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. Consider, for example, sequence numbers that increment once for every message a process receives. 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 [69, 78].

To leverage data independence, we need the application developers to supply us with a 'message fingerprint' function,[6] 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 [69]; 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 use this as a strategy to guide the choice of schedules, and can always fall back to exploring all schedules.

We combine observations #1 and #2 to pick a single, unique schedule as the initial schedule, 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 2, labeled "Initial Schedule".

Algorithm 1 Pseudocode for schedule exploration. We elide the details of DPOR for clarity (see [32] for a complete description). τ denotes the original execution; b.counterpart denotes the message delivery from τ that b is an alternate for; b.predecessors and b.sucessors denote the events before and after b when b was set.

```
procedure TEST(E')
  STSSCHED(E',τ,user defined fingerprint)
  if execution reproduced ✘ then return ✘
  while ∃_{b∈backtracks.} b's type=b.counterpart's type ∧
        b's fingerprint ≠ b.counterpart's fingerprint ∧
        time budget for E' not yet expired do
    prefix ← b.predecessors + b
    if prefix (or superstring) already executed then
      continue
    STSSCHED(E',prefix,user defined fingerprint)
    STSSCHED(E',b.successors,message type)
    if execution reproduced ✘ then return ✘
  return ✔
procedure STSSCHED(E',τ',match function)
  τ'' ← τ'.remove {e | e is external and e ∉ E'}
  for i from 0 to τ''.length do
    if τ''[i] is external then
      inject τ''[i]
    else if match(τ''[i]) ∈ pending then
      next ← pending.remove(match(τ''[i]))
      deliver next
      for e ∈ pending do
        if ¬commute(e,next) then
          set backtrack for e in place of next
```

Challenge: history-dependent message contents. This initial schedule can be remarkably effective, as demonstrated by the fact that minimization often produces good reduction even when we limit it to exploring this single schedule per external event subsequence. However, we find that the initial schedule fails to prune certain 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

[6] It may be possible to extract message fingerprints using program analysis or experimentation [17]. We have not yet explored those possibilities.

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 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 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 client command, it replicates it to the other members of the cluster 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.

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* implementation: the *type* of pending messages.

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 2. When searching for candidate schedules, if an internal message from the original execution is not pending at the point that internal message should be delivered, we skip it over. Similarly, we ignore any pending messages that do not match the corresponding type from the original execution. This leaves one remaining issue: how we handle cases where multiple pending messages match the corresponding original message's type.

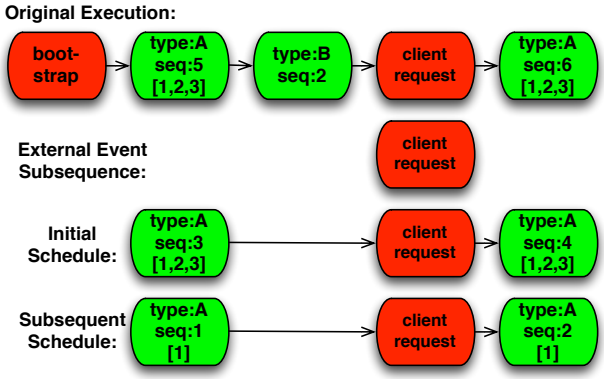

Figure 2: Example schedules. External message deliveries are shown in red, internal message deliveries in green. Pending messages, source addresses, and destination addresses are not shown. The 'B' message becomes absent when exploring a subsequence of external events. We choose an initial schedule that is close to the original, except for the fingerprinted 'seq' field. Next, we match only by type, allowing us to deliver pending messages with smaller contents.

Observation #4: Prioritize backtrack points that resolve match ambiguities. When there are multiple pending messages that match, we initially only pick one. DPOR 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 are most likely to be fruitful, because they keep the execution close to the causal structure of the original schedule except for small ambiguities in message contents.

Backtrack points allow us to explore all combinations of ambiguous pending messages. During this phase of minimization, note that we do not ignore the user-defined message fingerprint function entirely: we only prioritize backtrack points for pending messages that have the same type *and* that differ in their message fingerprints. We show the pseudocode for this phase as the while loop in Algorithm 1.

Minimizing internal events. Once delta debugging has completed, we attempt to further reduce the smallest reproducing schedule found so far. Here we apply the same minimization strategies, but instead of trying to prune external events, we try to remove internal events by simply not delivering them (leaving them pending). 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 we arrive at 1-minimal message contents.

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 to deprioritize certain message contents. To overcome inflation due to history-dependent message contents, we explore subsequent schedules by choosing backtrack points that match a more coarse-grained fingerprint: 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.

3.4 Comparison to Prior Work

We made observations #1 and #2 in our prior work [68]. 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 observation as 'STSSched', named after the 'STS' system where they were initially proposed [68].

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). Throughout the minimization process, we seek to spread a fixed time budget across each external event subsequence. We also seek to minimize internal events in addition to external events, and we shrink external message contents whenever possible (observation #5).

4 Implementation

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. On 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 [48], 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, but minimization could also be distributed across multiple machines.

Our interposition lies above the network transport layer; DEMi makes delivery decisions for application-level (non-segmented) messages. Since DEMi never passes messages to the transport layer, it may need to constrain the schedules it explores during fuzz testing and minimization, depending on the underlying network transport protocol the application assumes (TCP or UDP). In particular, if the application configures Akka to use TCP, DEMi must adhere to a FIFO delivery discipline between each source, destination pair.

Fuzz testing with DEMi. We begin by using DEMi to generate faulty executions. Developers give DEMi a test configuration, 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 the side-effects (e.g. by deleting disk contents) so that the environment is returned to its initial state at the end of each replay run.

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 running 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 the safety violation(s) it found,[7] a totally ordered recording of all events it executed, and additional 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. Although our minimization algorithms assume determinism, processes may behave non-deterministically. For our purposes, a process is non-deterministic if the messages it emits (modulo fingerprints) at a particular step 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 API calls. The Akka framework provides a narrow, general API that operates at a high level abstraction. This makes it easy for us to interpose. For example, Akka provides a timer API that obviates the need for developers to read directly from the system clock.

Nonetheless, applications may still be written to behave non-deterministically. In these cases, we add instrumentation (transparently with AspectJ wherever possible) to the application to mitigate the non-determinism.

akka-raft instrumentation. Within akka-raft, actors use a 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.

In addition to actors, Spark runs threads ('TaskRunners') that are outside the control of Akka. These threads execute the Spark job's computational tasks, and send status updates to other actors during their execution. The key challenge with threads that are outside the control of Akka is that during replay we do not know exactly when the messages we observed during the original execution will be resent by the threads, i.e. without knowing when the thread has completely processed the last message we sent it, we become dependent on wall-clock time to infer whether expected messages will or will not show up.

We manually add interposition points in two places within Spark's TaskRunners to avoid dependence on wall-clock time: the very start of the TaskRunner's execution, and the very end of the TaskRunner's execution. Before the TaskRunner starts, we signal to DEMi the identity of the TaskRunner. DEMi then records a 'start atomic block' event for that TaskRunner. During replay, if we are expecting a particular status update to be sent during the thread's execution, we have DEMi block until the 'end atomic block' before declaring the message as absent or not. This simple 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.

Other deviations from our computation model. Akka provides one other API that deviates from the computational model we defined in §2. Normally, actors only per-

[7] The recorded safety violation does not need to be specific to the exact state the system was in at the time of the violation. Less specific recorded safety violations are often better, since they allow DEMi to find divergent code paths that lead to the same buggy behavior.

form asynchronous tasks, e.g. sending messages to other actors with 'fire-and-forget' semantics. However, Akka also 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, 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 DEMi decides to deliver the response.

4.1 Limitations

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

Limited scale. Our current implementation of DEMi is tied to a single JVM, which may limit the scale of actor systems it can test. We do not believe this is fundamental.

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.

Framework dependence. Our tool is currently limited to applications built on top of the Akka framework.[8] If an application sends messages through some other channel besides Akka (e.g. a different RPC library), we would need to add additional interposition so that DEMi is aware of all message channels that affect control flow.

Shared memory & disk. In practice processes may communicate by writing to shared memory or disk rather than sending messages over the network. Although we do not currently support it, by adding interposition to the runtime system (as in [71]) we could treat writes to shared memory or disk in the same way that we treat network messages. More generally, adapting the basic DPOR algorithm to shared memory systems has been well studied [32, 81], and we could adopt these approaches.

Non-determinism. Mitigating non-determinism in the applications we evaluated required some amount of effort on our part. We might have adopted deterministic replay systems [29, 34, 54, 88] to avoid manual instrumentation. We did not primarily because we could not find a suitably supported record and replay system that operates at the right level of abstraction for Akka or other actor based systems. Note, however that deterministic replay by itself 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) is a more constrained problem than the one we solve, as discussed in §6.

[8] Interposing at a lower layer than the RPC library would allow DEMi to be applied more broadly. On the other hand, RPC layer interposition gives us finer grained control, access to application semantics, and a reduced schedule space size.

Bug Name	STSSched	TFB
raft-45	29s (275)	498s (6198)
raft-46	72s (448)	250s (4628)
raft-56	18s (233)	197s (3071)
raft-58a	137s (624)	43345s (834972)
raft-58b	27s (339)	42s (1747)
raft-42	118s (568)	10558s (176517)
raft-66	14s (192)	334s (10334)
spark-2294	330s (248)	97s (78)
spark-2294-caching	828s (467)	270s (179)
spark-3150	219s (174)	26s (21)
spark-9256	195s (154)	15s (12)

Table 2: Minimization runtime in seconds (total replays). Overall runtime is the summation of columns 2 and 3. Except for two case studies, our minimization techniques completed in under 20 minutes (1200s).

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 implementation achieves these properties simply by testing and minimizing on a single physical machine. However, obtaining a recording from a production system should not require coordination between machines; moreover, because the actor model only allows actors to process a single message at a time, we can record partial orders without vector clocks. Given all per-actor event logs, which record the order in which each actor processed received messages along with the set of messages they sent while processing each message, we can reconstruct the partial order of messages deliveries. Crash-stop failures do not need to be recorded, since from the perspective of other processes these are equivalent to 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 should be modest. For the systems we examined, Akka is primarily used as a control-plane, *not* a data-plane (e.g. Spark sends bulk data using Netty rather than Akka), where recording overhead is not particularly problematic.

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 minimization results in Table 1. The "Type" column shows two pieces of information: whether the bug can be triggered using TCP semantics (denoted as FIFO scheduling) 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 Name	Type	Initial	Provenance	STSSched	TFB
raft-45	Akka-FIFO, reproduced	1140 (E:108)	1129 (E:108)	545 (E:8)	37 (E:8)
raft-46	Akka-FIFO, reproduced	1730 (E:108)	1723 (E:108)	976 (E:9)	61 (E:8)
raft-56	Akka-FIFO, found	780 (E:108)	771 (E:108)	360 (E:8)	23 (E:8)
raft-58a	Akka-FIFO, found	2850 (E:108)	2824 (E:108)	953 (E:32)	226 (E:31)
raft-58b	Akka-FIFO, found	1500 (E:208)	1496 (E:208)	164 (E:13)	40 (E:9)
raft-42	Akka-FIFO, reproduced	1710 (E:208)	1695 (E:208)	1093 (E:39)	180 (E:21)
raft-66	Akka-UDP, found	400 (E:68)	392 (E:68)	262 (E:23)	77 (E:15)
spark-2294	Akka-FIFO, reproduced	1000 (E:30)	886 (E:30)	43 (E:3)	40 (E:3)
spark-2294-caching	Akka-FIFO, reproduced	700 (E:3)	667 (E:3)	64 (E:3)	51 (E:3)
spark-3150	Akka-FIFO, reproduced	600 (E:20)	536 (E:20)	18 (E:3)	14 (E:3)
spark-9256	Akka-FIFO, found (rare)	600 (E:20)	541 (E:20)	16 (E:3)	16 (E:3)

Table 1: Overview of case studies. "E:" is short for "Externals:". Column 5 subtracted from column 3 shows how much overall reduction our techniques achieve; subtracting column 4 shows how much reduction can be achieved statically; subtracting column 5 from column 4 shows how our new techniques compare to the state of the art [68].

bug.[9] The "Provenance" column shows how many events from the initial execution can be statically pruned by computing which of them are concurrent with the safety violation. The STSSched column shows how much reduction DEMi achieves after checking the initial schedules prescribed by our prior work [68]. TFB shows the additional minimization our techniques ('Type Fingerprints with Backtracks') provide, where we direct DPOR to explore as many backtrack points that resolve type ambiguities (but no other backtrack points) as possible within the 16 hour time budget we provided. Overall we find that DEMi is able to achieve as much as 97% trace reduction, and provides up to 16 times smaller results than our previous techniques (STSSched). Replayable executions for all of these case studies are publicly available at github.com/NetSys/demi-experiments.

We create the initial executions for all of our case studies by generating fuzz tests with DEMi and selecting the first fuzz test that triggers the invariant violation with >350 initial message deliveries.[10] 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 (e.g., we tried to avoid fuzz sequences that would be highly unlikely to be encountered in practice).

As noted in the introduction, the systems we focus on are: akka-raft [3], an existing implementation of the Raft consensus protocol, and Apache Spark [86], a widely used large scale data analytics framework. akka-raft, as an early-stage software project, demonstrates how DEMi (and fuzz testing more generally) can be used to aid the development process. Our evaluation of Spark demonstrates that DEMi can be applied to complex, large scale distributed systems.

Reproducing Sequence Size. Ideally, we would like to show how far our minimization gains are from the perfectly optimal result, e.g., a result computed by exhaustively enumerating the schedule space for a given bug. Unfortunately, enumerating the entire schedule space is intractable in most cases, and often even impossible.[11]

We instead compare the minimality of a sampling of our case studies against the smallest fault-inducing executions we could construct by hand. For some case studies, the minimized result was very close to the smallest trace we could produce by hand (e.g., the smallest reproducing trace we could produce by hand for raft-56 was 20 events, whereas DEMi found a 23 event sequence). For others, e.g., raft-58a, the execution was very clearly inflated compared to the subsequence ($\sim$ 30 events) constructed by us. In this case, DEMi exhausted its time budget while checking smaller external event subsequences.

Minimization Pace. To measure how quickly DEMi makes progress, we graph its progress as a function of the number of schedules it executes. Figure 3 shows an example for the raft-58b case study. All of the other case studies follow the same general pattern, where significant progress is made early on, then only rare progress is made.

We also show how much time (# of replays) DEMi took to reach completion of STSSched (initial schedules) and TFB (where only backtrack points that resolve type ambiguities are explored) in Table 2. The time budget we allotted to DEMi for all case studies was 16 hours (57600s). All case studies except raft-58a reached completion of TFB before the time budget expired. For raft-58a, the time budget was spread roughly

[9] It is possible but highly unlikely to trigger spark-9256 in practice. Nonetheless, it is interesting as an evaluation of minimization.

[10] Note that because initial trace size can be arbitrarily inflated, trace reduction is not a robust metric. The more important evaluation question is whether DEMi is able to consistently produce small results, regardless of initial trace size.

[11] Any asynchronous distributed system that requires reliable communication (i.e. acknowledgment from peers that messages have been received) is not guaranteed to stop sending messages [9], essentially because nodes cannot distinguish between crashes of their peers and indefinite message delays.

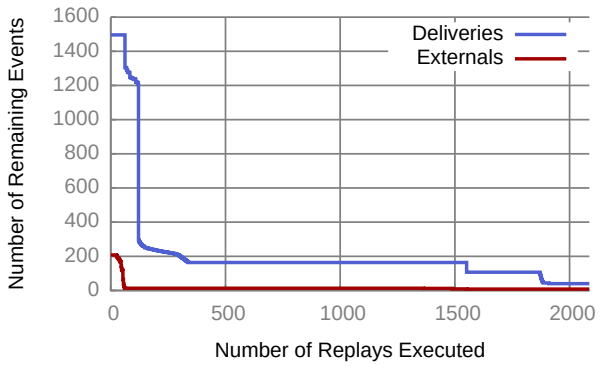

Figure 3: Minimization pace for raft-58b. Significant progress is made early on, then progress becomes rare.

evenly across all external event subsequences.

When reading the runtime results in Table 2 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 it interposes on timer events, and can trigger timer events before the wall-clock duration for those timers has actually passed, without the application being aware of this fact (cf. [37]).

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 [38]. Nonetheless, since humans can only keep a small number of facts in working memory [60], minimization seems generally useful. As one developer puts it, "Automatically shrinking test cases to the minimal case is immensely helpful" [73].

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 yet been deployed in production. There were no existing tests or bug fixes for any of the known bugs we reproduced (raft-42, raft-45, raft-46); these were found by a manual audit of the code.

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. 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 [63]: 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 invariant (State Machine Safety).

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 we found would have been infeasible to anticipate a priori. Second, debugging unminimized faulty executions would be very time consuming and conceptually challenging; the most fruitful debugging methodology we found was to walk through events one-by-one, which would take hours for unminimized executions.

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 manually walked through the sequence of message deliveries in the minimized execution, one at a time. 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 the places in the execution that deviate from our understanding of correct behavior. We then examined the akka-raft code to understand why it deviated, and came up with a fix. Finally, we reran the recorded execution after the bug fix to see if the bug still surfaced.

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 bugs from previous case studies before moving on to the next case study. We reported all bugs and fixes to the akka-raft developers.

Due to space limitations we describe only the most interesting case studies here. For full descriptions of each case study, click the hyperlinks in Table 1.

raft-56: Nodes forget who they voted for. akka-raft is written as a finite state machine. 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-42: Quorum computed incorrectly. Leaders are supposed to commit log entries to their state machine when they knows that a quorum (N/2+1) of the pro-

cesses 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 N/2+1st 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. 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.

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., a single, statically defined master, directing a fleet of worker nodes).

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.

Our main takeaway from Spark is that, for *simple* Spark jobs, STSSched does surprisingly well. From examining the execution traces we believe this is because Spark's communication tasks are all almost entirely independent of each other. However, for more complex Spark jobs, particularly those that *cache* intermediate result to be used as inputs to later stages, STSSched does not minimize as effectively (as demonstrated in the spark-2294-caching case study). This is because the messages Spark sends to track the locations of cached intermediate results are *history-dependent*.

spark-3150: Simultaneous failure causes infinite restart loop. Spark's master node supports a 'Cold-Replication' 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.

In 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.

	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: Results of external message shrinking for raft-45 starting with 9 processes. Message shrinking + minimization was able to reduce the cluster size to 5 nodes.

	akka-raft	Spark
Message Fingerprint	59	56
Non-Determinism	2	∼250

Table 4: Instrumentation complexity (lines of code) needed to define message fingerprints, and to mitigate non-determinism.

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 show the benefits of shrinking the contents of bootstrapping messages (cluster size) on a 9 node akka-raft cluster, where we trigger the raft-45 bug with a fuzz test. We first shrank message contents as much as possible 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 and to mitigate non-determinism with the application modifications described in §4. For the non-determinism mitigation row, we exclude our AspectJ interposition on the Akka APIs.

6 Related Work

We start our discussion of related work with the most closely related literature.

Input Minimization for Sequential Programs. Minimization algorithms for sequentially processed inputs are well-studied [19, 20, 24, 38, 67, 77, 90]. These are not immediately applicable to inputs that are interleaved with internal events of concurrent processes.

Best-Effort Minimization for Concurrent Systems. Several tools minimize inputs to concurrent systems in a *best-effort* fashion [11, 26, 42, 45, 74], but none exercise control over sources of non-determinism. We found

in our previous work [68] that minimization makes little progress without interposition. At best, these approaches replay each subsequence multiple times and hope that the violation is reproduced at least once [25, 42].

QuickCheck's PULSE controls the message delivery schedule [25], but its 'shrinking' strategy is still best-effort: it simply skips any expected messages that are not pending when checking a shrunk schedule [41].

Thread Schedule Minimization. Other techniques seek to minimize thread interleavings leading up to concurrency bugs [22, 30, 39, 44]. 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 [40,52,71]. These techniques do not explore alternate code paths that still trigger the invariant violation. Program analysis also over-approximates state reachability, disallowing them from removing dependencies that actually semantically commute.

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

Abstract model checkers convert (possibly concurrent) programs to logical formulas, find logical contradictions (invariant violations) using solvers, and minimize the logical conjunctions to aid understanding [23, 47, 59]. These are tied to a single language and typically assume access to all source code including libraries, 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 [12]. 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 (cf. [6]), and 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 [18, 28, 33, 53, 55, 61, 62, 64, 72, 75, 80]. We do the same, but crucially, we are able to use information from previously failing executions to guide our search. We are also the first to observe that the application's use of TCP constrains the schedule space.

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 [83, 84] and to apply heuristics for choosing initial schedules [51], but these changes are intended to help *find* bugs.

Bug Reproduction. Several papers seek to find a schedule that reproduces a given concurrency bug [10, 65, 87, 88]. 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 [13,21,46,66,85]. We find exact reproducing scenarios, and currently focus on improving the productivity of early stage development rather than production debugging.

Log Comprehension. Model inference techniques summarize log files in order to make them more easily understandable by humans [14–16,56,57]. Model inference does not modify the event logs.

Program Slicing & Automated Debugging. Program slicing [76] and the subsequent literature on automated debugging [27, 43, 58, 70, 79, 91] 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 systems are lagging behind their sequential counterparts. Inspired by the obvious utility of input minimization tools, we sought to develop a minimization tool for distributed executions. Our evaluation results for two very different distributed systems leave us optimistic that these techniques can be successfully applied to a wide range of distributed 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] Generating a Random Program vs. Generating All Programs. http://blog.regehr.org/archives/1246.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[36] 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.

[37] 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.

[38] 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.

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

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

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

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

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

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

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

[46] 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.

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

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

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

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

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

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

[53] 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.

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

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

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

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

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

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

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

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

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

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

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

[65] 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.

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

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

[68] 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.

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

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

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

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

[73] A. Thompson. http://tinyurl.com/qgc387k.

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

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

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

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

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

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

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

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

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

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

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

[85] 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.

[86] 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.

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

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

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

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

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