**Title**
Reducing Faulty Executions of Distributed Systems



**Author**
Scott, Robert Colin Butler

**Publication Date**
2016

Peer reviewed|Thesis/dissertation

**Reducing Faulty Executions of Distributed Systems**

by

Robert Colin Butler Scott

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Scott Shenker, Chair
Professor George Necula
Professor Sylvia Ratnasamy
Professor Tapan Parikh

Summer 2016

**Reducing Faulty Executions of Distributed Systems**

## Abstract

Reducing Faulty Executions of Distributed Systems

by

Robert Colin Butler Scott

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Chair

When confronted with a buggy execution of a distributed system—which are commonplace for distributed systems software—understanding what went wrong requires significant expertise, time, and luck. As the first step towards fixing the underlying bug, software developers typically start debugging by manually separating out events that are responsible for triggering the bug (signal) from those that are extraneous (noise).

In this thesis, we investigate whether it is possible to automate this separation process. Our aim is to reduce time and effort spent on troubleshooting, and we do so by eliminating events from buggy executions that are not causally related to the bug, ideally producing a "minimal causal sequence" (MCS) of triggering events.

We show that the general problem of execution minimization is intractable, but we develop, formalize, and empirically validate a set of heuristics—for both partially instrumented code, and completely instrumented code—that prove effective at reducing execution size to within a factor of 4.6X of minimal within a bounded time budget of 12 hours. To validate our heuristics, we relay our experiences applying our execution reduction tools to 7 different open source distributed systems.

To my parents

# Contents

# List of Figures

# List of Tables

## Acknowledgments

I would like to acknowledge my advisors: Scott Shenker, Arvind Krishnamurthy, and recently, George Necula. Scott's humility was the main reason I chose to come to Berkeley for graduate school. He taught me, among many things, how to transform my stilted writing and speaking into engaging story lines, and how to identify the simplicity at the core of complex technical concepts. It is thanks to his extraordinary support that I was able to move to Seattle midway through graduate school. Arvind is responsible for letting me out into the academic world, and he took me under his wing again when I returned to Seattle. His optimism and excitement are absolutely infectious. His insistence on surrounding oneself with *nice* collaborators has served me well throughout my career. George is the most genuine academic I have ever met. He does not, as far I can tell, 'play the game' nor seek to heighten his own reputation; he seems to be motivated purely by a search for truth and value. Although I will likely never match his raw intellect, it would be an admirable achievement to adopt some of his attitudes.

I am also indebted to many other collaborators who have helped shaped the content of this dissertation. First and foremost, I am indebted to my officemate and friend Aurojit Panda. Panda's knowledge, both technical and non-technical, is astounding. His role in shaping both of the core chapters of this dissertation has been invaluable, and the breadth of topics I learned about through my conversations with him came in handy on several occasions in my 1:1 job interviews. Andreas Wundsman taught me a great deal about how to write effective, appropriately structured code, and how to best "debug the debugger". Without his aid I am not sure I would have persevered through my third year of graduate school. Sam Whitlock, Ahmed El-Hassany, Vjekoslav Brajkovic, Zhi Liu, Andrew Or, Jefferson Lai, Eugene Huang, Kyriakos Zarifis, and Barath Raghavan also played an instrumental role in helping me shape the ideas and results presented here.

Vern Paxson, Ali Ghodsi, and Barath Raghavan, each in their own way, taught me the value and importance of focusing on details. My competency as a researcher has been improved by their urging.

When I first met Ethan Katz-Bassett, I did not yet know that 'dictionary' is synonym for 'hashmap', and when he gave me instructions for running my first experiment, I mistakenly believed that he was asking me to take out pen and paper and painstakingly count IP addresses by hand. Thanks to his mentorship, I was able to successfully start along the path to where I am today. In my own work with undergraduates I have tried to emulate his innate mentorship abilities.

My perspective on academia and its role in the world has been shaped by my multiple experiences as a software engineering intern. Michael Piatek, Michael Buettner, and Teemu Koponen served as great mentors and role models during those internships.

Among my graduate student peers I have found some of my dearest friends. Justine Sherry has provided advice, example-setting, and rapport since my junior year of college. I shared many laughs with my Berkeley officemates Radhika Mittal, Kay Ousterhout, Amin Tootoonchian, Murphy Mccauley, Ethan Jackson, Qifan Pu, Wenting Zheng, and (de facto) Shivaram Venkataraman, as well as my UW officemates Will Scott, Qiao Zhang, Danyang Zhou, Vincent Liu, Haichen Shen, Seungyeop Han, Donny Huang, Ray Cheng, and Sophia Wang. Sangjin Han has perhaps the highest signal (insight) to noise (gossip) ratios of anyone I know. Peter Bailis, through his example, taught me the value of promoting one's research, especially to non-academic audiences. I will never forget my housemates, especially Austin Buchan, Armin Samii, Ali Köksal, and Martin Maas.

I would be remiss not to thank Maria Clow, who continues to teach me to be a better person. I have cherished her companionship through a wide range of experiences, even a Category 4 hurricane.

Lastly, I am grateful to my family for their love and support.

# Chapter 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. The number of event orderings a distributed system may encounter due to non-determinism grows exponentially with the number of events in the execution. Bugs are therefore commonplace in distributed systems software, since developers invariably overlook some unanticipated event orderings amid the exponential number of orderings they need to consider.

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 randomized concurrency testing (where sequences of message deliveries, failures, etc. are injected randomly 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 *reduced*, so that only events that are relevant to triggering the bug remain. In fact, software developers often start troubleshooting by manually performing this reduction; they identify (i) which events in the execution caused their system to arrive at the unsafe state, and (ii)

[1] Developers spend a significant portion of their time debugging (49% of their time according to one study [69]), especially when the bugs involve concurrency (70% of reported concurrency bugs in [43] took days to months to fix).

which events are irrelevant (and can be ignored).

Since developer time is typically much more costly than machine time, automated reduction tools for *sequential* test cases [24, 136, 127] have already proven themselves valuable, and are routinely applied to bug reports for software projects such as Firefox [1], LLVM [79], and GCC [42]. In this dissertation, we argue the following:

> **Thesis statement:** *It is possible to automatically reduce faulty executions of distributed systems.*

We primarily focus on faulty executions generated in an automated test environment. However, we also illustrate how one might reduce production executions.

We seek to provide execution reduction tools that leave developers of distributed systems with a 'minimal causal sequence' of triggering events, which are necessary and sufficient for triggering the bug. We claim that the greatly reduced size of the trace makes it easier for the developer to figure out which code path contains the underlying bug, allowing them to focus their effort on the task of fixing the problematic code itself. As far as we know, we are the first to show how to provide execution reduction tools for distributed and concurrent systems without needing to analyze the code.

## 1.1 Challenges

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. Execution reduction algorithms must therefore consider both which input events and which (of the exponentially many) event interleavings ("schedules") are likely to be necessary and sufficient for triggering the bug. Our main contribution throughout this dissertation 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 wall-clock timers to detect failures). To reduce executions consistently and reliably, we need to find ways to cope with this non-determinism.

We address these challenges from two perspectives.

### 1.1.1 Execution Reduction without Application-Layer Interposition

In chapter 3, we pose the following question:

*Can we automatically reduce executions without making any assumptions about the language or instrumentation of the software under test?*

The key advantage of this 'blackbox' approach to execution reduction is that it allows us to avoid the (often substantial) engineering effort required to instrument and completely control all the sources of non-determinism in distributed systems. A blackbox execution reduction tool can be applied easily to a wide range of systems without much effort.

Rather than analyzing the code (which would violate our blackbox requirement), we *experiment* with different executions and observe how the system behaves as a result. If we find an execution that is shorter than the original and still triggers the same bug, we can then ignore any of the events that we did not include from the original execution. We continue experimenting with shorter executions until we know that we cannot make further progress on reducing the size of the execution.

We develop our blackbox execution reduction techniques in the context of software-defined networking (SDN) control software (explained in chapter 2.3). In lieu of application-layer interposition, we interpose on a standard protocol used by all SDN control software: OpenFlow [87].

We develop an intuitive heuristic for choosing which event interleavings to explore: we know that the original execution triggered the bug, so, when exploring modified executions we should try to stay as close as possible to the original execution's causal dependencies. The mechanism we develop to implement this heuristic involves dynamically inferring whether events we expect to show up (from the original execution) will in fact show up in modified executions.

Without perfect interposition, we need to find ways to mitigate the effects of non-determinism while we experimentally explore event interleavings. Through case studies, we discover an effective set of mitigation strategies involving careful wall-clock spacing between events, and replaying non-deterministic executions multiple times.

### 1.1.2 Execution Reduction with Application-Layer Interposition

Our investigation in chapter 3 focuses narrowly on one kind of distributed system, and, without complete control over the execution, the heuristics we develop there leave room for improvement. In chapter 4, we identify a computational model that allows us to cleanly reason about new reduction heuristics. We also apply these heuristics successfully to several types of distributed systems besides SDN control software.

The computational model we identify—the actor model—encapsulates all sources of non-determinism from the network in a small number of interposition points. This makes

it easy for us to gain (nearly) perfect control over the execution of events in actor-based distributed systems. With this control in place, it is theoretically possible to simply enumerate all possible schedules, execute each one, and pick the smallest execution that triggers the bug. The space of all possible schedules is intractably large however, which leads to our main question:

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

Our approach is to carefully prioritize the order in which we explore the schedule space. We prioritize schedules that are shorter than the original, and that we believe will be likely to retrigger the invariant violation (since retriggering the invariant violation allows us to make progress on reducing the execution).

For any prioritization function we choose, an adversary could construct the program under test, or even just the initial execution, so that our our prioritization function does not make any reduction progress within a limited time budget.

Fortunately, the systems we care about in practice are not constructed by adversaries. They exhibit certain *program properties*, or constraints on their computational structure, that we can take advantage of when designing our prioritization functions. We choose our prioritization functions based on our understanding of these program properties.

Drawing from what we learned in chapter 3, the main program property we leverage states that if one schedule triggers a bug, schedules that are "similar" in their causal structure should have a high probability of also triggering the bug. 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 develop a hierarchy of equivalence relations between events, and show that systematically exploring schedules that are close to the original execution yield significant gains in reduction over the previous heuristics described in chapter 3. We implement these heuristics and use them to find and reduce several bugs in two very different types of distributed systems: the Raft consensus protocol and the Spark data analytics engine.

## 1.2 Outline and Previously Published Work

The remainder of this dissertation proceeds as follows. We provide background on the existing test case reduction algorithms we build upon in chapter 2. In chapter 3 we design and evaluate execution reduction strategies that do not assume application-layer interposition. In chapter 4 we identify a deterministic (yet practically relevant) computational

model for distributed systems, and show how to effectively reduce executions of systems that adhere to that more refined model. We discuss how both of these approaches relate to previous literature in chapter 5, and we comment on avenues for future work and conclude in chapter 6.

Chapter 3 revises published material from [104]. Chapter 4 revises published material from [103].

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

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

$$ddmin_2(T'_{\text{✘}}, R) = \begin{cases} T'_{\text{✘}} & \text{if } |T'_{\text{✘}}| = 1 \text{ (“base case”)} \\ ddmin_2(T_1, R) & \text{else if } test(T_1 \cup R) = \text{✘} \text{ (“in } T_1\text{”)} \\ ddmin_2(T_2, R) & \text{else if } test(T_2 \cup R) = \text{✘} \text{ (“in } T_2\text{”)} \\ ddmin_2(T_1, T_2 \cup R) \cup ddmin_2(T_2, T_1 \cup R) & \text{otherwise (“interference”)} \end{cases}$$

where ✘ denotes an invariant violation, $T_1 \sqsubset T'_{\text{✘}}$, $T_2 \sqsubset T'_{\text{✘}}$, $T_1 \cup T_2 = T'_{\text{✘}}$, $T_1 \cap T_2 = \emptyset$, and $|T_1| \approx |T_2| \approx |T'_{\text{✘}}|/2$ hold.

Figure 2.1: Delta Debugging Algorithm from [135]. $\sqsubseteq$ and $\sqsubset$ denote subsequence relations. TEST is defined in Algorithm 2.

# Chapter 2

# Background Material

## 2.1 Delta Debugging

Delta debugging is an algorithm for reducing test cases [135, 136]. The inputs to delta debugging are a test input (e.g., an HTML file) that triggers a bug, a function for separating out the components of the test input (e.g. individual HTML tags, or individual ASCII characters), and a test runner (e.g., a script that starts a browser process, feeds in the test HTML page to the browser, and finally checks whether the browser exhibited a bug or not). Delta debugging seeks to reduce the size of the test input to a “minimal” counterexample (a subset of the original test case) that still triggers the bug.[1]

[1]By “minimal”, we do not mean global minimality, since finding a globally minimal subset is intractable (it is equivalent to enumerating the powerset of the original test case). Delta debugging instead seeks to produce a 1-minimal subset, meaning that no single component of the subset can be removed while still triggering the bug.

| Step | Input Components | | | | | | | | 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 2.1: Example execution of Delta Debugging, taken from [135]. '·' denotes an excluded input component (external event), ✘ denotes an invariant violation, and ✔ denotes lack of an invariant violation.

Delta debugging works by experimenting with different subsets of the original test case. Each experiment involves feeding a subset of the original test case to the test runner and checking whether the bug is still triggered. If the bug is still triggered, we know that the components of the original test case that are not part of the subset are not necessary for triggering the bug, and we can therefore ignore them. Delta debugging continues experimenting with subsets until it knows that it has reached a 1-minimal result.

The simplest approach for selecting subsets of the original test case would be to remove one component at a time. Delta debugging does better in the average case by splitting the components in a fashion similar to binary search. We show a detailed specification of the delta debugging simplification algorithm (the simple version [135] that we use in this paper) in Figure 2.1.

For illustration, consider the execution of delta debugging shown in Table 2.1. The original test case is a sequence of components $e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8$ that, when executed by the test runner, results in an invariant violation ✘. Delta debugging first checks the left half of the components. The bug is not triggered (as reported by the test runner 'TEST'), so delta debugging then tries the right half of the components. Again, the bug is not triggered, so delta debugging recurses, splitting the components into fourths instead of halves. Whenever the invariant violation ✘ is triggered, delta debugging knows that the components not included in that subset can be ignored from then on. Delta debugging continues trying to find which components are not necessary for triggering the invariant violation. After the 7th experiment, delta debugging knows that it has reached a 1-minimal result, and produces the output $e_3, e_6$.

To be guaranteed to produce 1-minimal outputs, the version of delta debugging that we employ [135] makes three assumptions about inputs: monotonicity, unambiguity, and consistency. Inputs that violate monotonicity may contain components that "undo" the

invariant violation triggered by the MCS, and may therefore exhibit inflated MCSes. Inputs that violate unambiguity may exhibit multiple MCSes; delta debugging will return one of them. The most important assumption is consistency, which requires that the test outcome can always be determined, i.e., that each subset chosen by delta debugging is semantically valid.

In this dissertation, we guarantee neither monotonicity nor unambiguity. We do however codify domain knowledge in order to ensure that each subset chosen by delta debugging is semantically valid. By guaranteeing consistency, we can use a simple version of delta debugging [135] that does not consider subset complements. The version of delta debugging that considers subset complements [136] has the advantage that it is guaranteed to produce 1-minimal output even in the face of inconsistency, but it incurs an additional factor of n in worst-case runtime complexity.

The delta debugging algorithm we use terminates in $\Omega(\log n)$ invocations of the test runner in the best case, and $O(n)$ invocations in the worst case, where $n$ is the number of inputs in the original trace [135]. If it is difficult to ensure consistency, one could employ the more expensive version of delta debugging, which terminates in $O(n^2)$ invocations in the worst case [136].

## 2.2 Dynamic Partial Order Reduction

Delta debugging is designed for sequential test cases. In a concurrent environment, the behavior of a software system depends not only on its inputs (e.g., commands from the user or failure events), but also on interleavings of internal events (e.g., the order of reads and writes to shared memory, as determined by a thread scheduler). To properly reduce executions of concurrent systems, we therefore need to consider both the input events (which can be considered together as a sequential collection of components to be reduced by delta debugging), and the interleavings of internal events.

The key source of concurrency in a distributed system is the network. In response to external (input) events, processes send messages to each other over the network, which may (in the case of a fully asynchronous network) be arbitrarily reordered or delayed by the network before they are delivered to their destination.

If we precisely control when the network delivers each message (as we do in chapter 4), we can systematically explore the space of possible event interleavings. For a given sequence of input (external) events, there is a finite number of possible event interleavings (assuming a maximum bound on the number of message deliveries in the execution). We could thus enumerate every possible event interleaving for each input subset chosen by delta debugging, and report back to delta debugging whether any one of those interleav-

**Algorithm 1** The original depth-first version of Dynamic Partial Order Reduction from [35]. $last(S)$ denotes the configuration reached after executing $S$; $next(\kappa, m)$ denotes the state transition (message delivery) where the message m is processed in configuration $\kappa$; $\rightarrow_S$ 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(∅)
procedure EXPLORE(S)
    κ ← last(S)
    for each message m ∈ pending(κ) do
        if ∃i = max({i ∈ dom(S) | S_i is dependent and may be coenabled with next(κ, m) and i ↛_S m}):
            E ← {m' ∈ enabled(pre(S, i)) | m' = m or ∃j ∈ dom(S) : j > i and m' = msg(S_j) and j →_S m}
            if E ≠ ∅:
                add any m' ∈ E to backtrack(pre(S, i))
            else
                add all m ∈ enabled(pre(S, i)) to backtrack(pre(S, i))
    if ∃m ∈ enabled(κ):
        backtrack(κ) ← {m}
        done ← ∅
        while ∃m ∈ (backtrack(κ) \ done) do
            add m to done
            EXPLORE(S.next(κ, m))
```

ings triggered the invariant violation. Unfortunately, this approach is wildly intractable: the number of possible event interleavings is factorial in the number of events.

As others have observed [44], many events occurring in a schedule (event interleaving) 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 $a$ to be delivered to process $c$, and $e_2$ is a message from process $b$ to be 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 [68]: 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.[2]

Partial order reduction (POR) [44, 35] is a well-studied technique for pruning commutative schedules from the search space. In the above example, given two schedules that

[2]Stronger forms of commutativity may hold if events cannot possibly be causally unrelated to each other, for example when a distributed algorithm is stateless. Inferring such cases of commutativity would require understanding of application semantics; in contrast, happens-before commutativity is independent of the application.

only differ in the order in which $e_1$ and $e_2$ appear, POR would only explore one schedule. Dynamic POR (DPOR) [35] is a refinement of POR: 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.

We show the original depth-first version of Dynamic Partial Order Reduction in Algorithm 1. We use DPOR as the basis for our execution reduction strategies in chapter 4; we modify it to explore the schedule space in a prioritized order.

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, to be explained later in chapter 4). 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 well before we run out of memory.

There are a few desirable properties of the unmodified DPOR algorithm that 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.

**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.

## 2.3 Software Defined Networking

In chapter 3 we evaluate our blackbox execution reduction techniques on software-defined networking (SDN) control software. Here we provide background on the SDN architecture, and the types of bugs one might encounter in an SDN network.

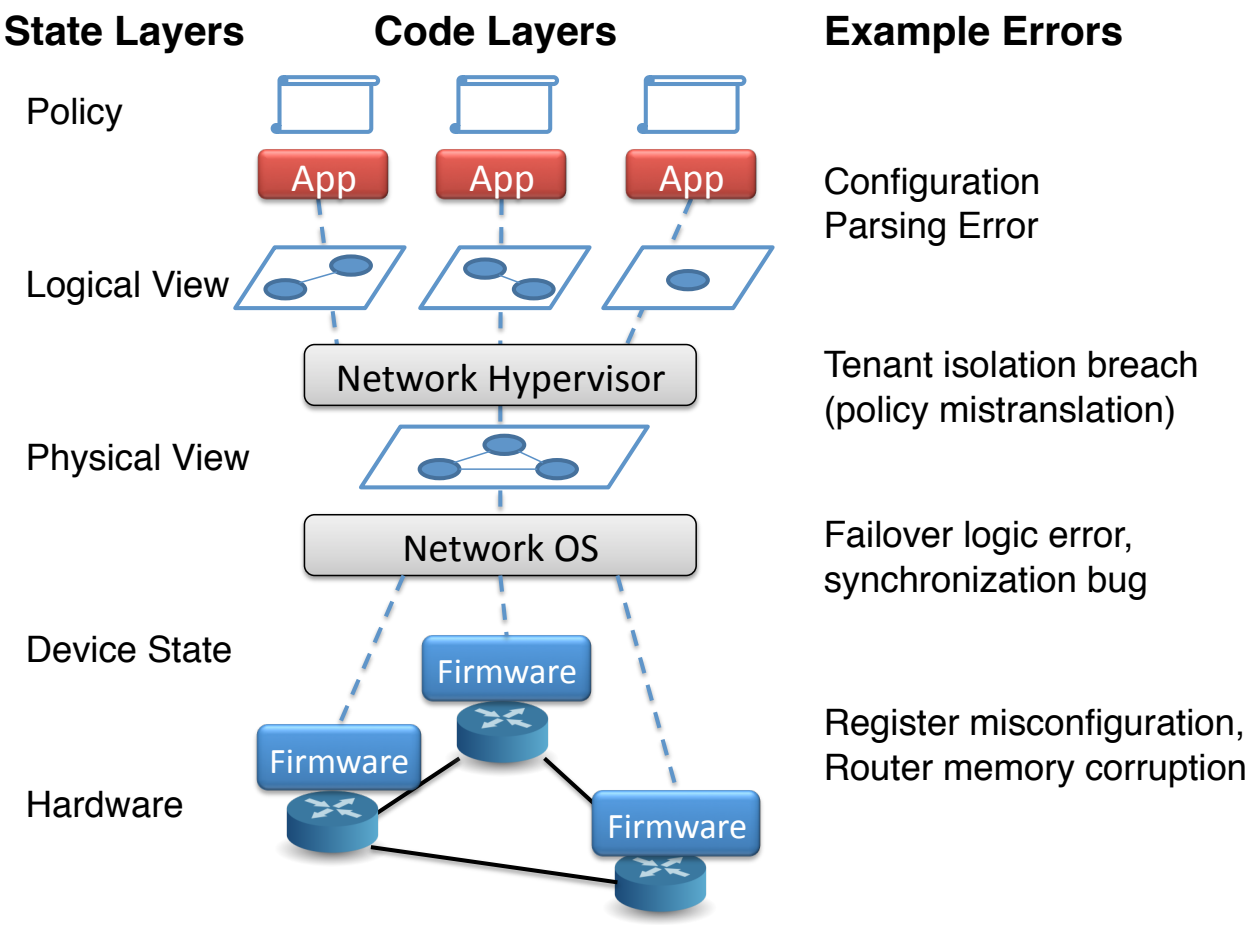



Figure 2.2: The SDN Stack: state layers are on the left, code layers are in the center, and example bugs are on the right. Switches and routers are at the bottom layer of the stack.

SDN is a type of network architecture, designed to simplify network management. It achieves simplicity by factoring the control plane of the network into a hierarchy of abstractions. At the highest layer of abstraction, network operators specify policies for the network. Layers beneath translate these high-level policies into low-level, policy-compliant packet forwarding behavior.

We depict the layers of the SDN control plane in Figure 2.2. *State layers* hold a representation of the network's configuration, while *code layers* implement logic to maintain the mapping between two state layers. State changes propagate in two directions: policy changes from above map to configurations (i.e., forwarding entries) of lower layers, while network events from below, such as link failures, map to view changes above.

At the highest layer are "control applications" (e.g., OpenStack Quantum [93]) that specify routing, access control, or quality of service policies by configuring the logical view. The logical view is a simplified, abstract representation of the network (often a

single logical switch) designed to make it easy for the control application to specify policies. The network hypervisor maps the configuration of the logical entities to one or more physical entities in the physical view. The physical view has a one-to-one correspondence with the physical network, and it is the job of the network operating system (e.g., Onix [66], NOX [46]) to configure the corresponding network devices through a protocol such as OpenFlow [87]. Finally, the firmware in each network device maps forwarding tables to hardware configuration.

A key observation is that the *purpose* of the entire architecture is to translate human intent to the low-level behavior of the network, and each layer performs one piece of this translation process. Therefore, at any point in time, every state layer should be correctly mapped to every other state layer, a property we loosely call "equivalence".

*Errors within the SDN control stack always result in broken equivalence between two or more layers.* For example, a breach of tenant isolation manifests itself as an inconsistency between a policy specified at the logical view ("Network A should not be able to see network B's packets") and the state of the physical network ("One of the switches is sending packets to the wrong destination network"). If all state layers are equivalent, but unwanted behavior persists, it must be the case that the configured policy does not match the operator's intent, or hardware beyond the control plane's view, such as an optical repeater, is broken.

When the behavior of the network does not match an operator's intent, the operator must localize the symptoms to the system components that are responsible. Then, the operator needs to understand what events (which determine the code path that leads up to the problem) caused that component to misbehave. In this dissertation we focus on making it easier for the operator to understand the events caused the component to misbehave.

# Chapter 3

# Reducing Faulty Executions of SDN Control Software, Without Application-Layer Interposition

## 3.1 Introduction

Software-defined networking (SDN) proposes to simplify network management by providing a simple logically-centralized API upon which network management programs can be written. However, the software used to support this API is anything but simple: the SDN control plane (consisting of the network operating system and higher layers) is a complicated distributed system that must react quickly and correctly to failures, host migrations, policy-configuration changes and other events. All complicated distributed systems are prone to bugs, and from our first-hand familiarity with five open source controllers and three major commercial controllers we can attest that SDN is no exception.

When faced with symptoms of a network problem (*e.g.* a persistent loop) that suggest the presence of a bug in the control plane software, software developers need to identify which events are triggering this apparent bug before they can begin to isolate and fix it. This act of "troubleshooting" (which precedes the act of debugging the code) is highly time-consuming, as developers spend hours poring over multigigabyte execution traces.[1] Our aim is to reduce effort spent on troubleshooting distributed systems like SDN control software, by automatically eliminating events from buggy traces that are not causally related to the bug. We seek to produce a "minimal causal sequence" (MCS) of triggering events.

[1]Software developers in general spend roughly half (49% according to one study [43]) of their time troubleshooting and debugging, and spend considerable time troubleshooting bugs that are difficult to trigger (70% of reported concurrency bugs in [43] took days to months to fix).

Our goal of reducing execution traces is in the spirit of delta debugging [135], but our problem is complicated by the distributed nature of control software: our input is not a single file fed to a single point of execution, but an ongoing sequence of events involving multiple actors. We therefore need to carefully control the interleaving of events in the face of asynchrony, concurrency and non-determinism in order to reproduce bugs throughout the reduction process. In contrast to deterministic replay approaches, we strive to explore divergent execution paths. Crucially, in this chapter, we aim to reduce execution traces without making assumptions about the language or instrumentation of the control software.

In this chapter we demonstrate a troubleshooting system that, as far as we know, is the first to meet these challenges (as we discuss further in chapter 5). Once our system reduces a given execution trace to an MCS (or an approximation thereof), the developer embarks on the debugging process. We claim that the greatly reduced size of the trace makes it easier for the developer to figure out which code path contains the underlying bug, allowing them to focus their effort on the task of fixing the problematic code itself. After the bug has been fixed, the MCS can serve as a test case to prevent regression, and can help identify redundant bug reports where the MCSes are the same.

Our troubleshooting system, which we call STS (SDN Troubleshooting System), consists of 23,000 lines of Python, and is designed so that organizations can implement the technology within their existing QA infrastructure (discussed in chapter 3.5); we have worked with a commercial SDN company to integrate STS. We evaluate STS in two ways. First and most significantly, we use STS to troubleshoot seven previously unknown bugs—involving concurrent events, faulty failover logic, broken state machines, and deadlock in a distributed database—that we found by fuzz testing five controllers (Floodlight [36], NOX [46], POX [86], Pyretic [40], ONOS [92]) written in three different languages (Java, C++, Python). Second, we demonstrate the boundaries of where STS works well by finding MCSes for previously known and synthetic bugs that span a range of bug types. In our evaluation, we quantitatively show that STS is able to reduce (non-synthetic) bug traces by up to 98%, and we anecdotally found that reducing traces to MCSes made it easy to understand their root causes.

## 3.2 Background

Network operating systems, the key component of SDN software infrastructure, consist of control software running on a replicated set of servers, each running a controller instance. Controllers coordinate between themselves, and receive input events (*e.g.* link failure notifications) and statistics from switches (either physical or virtual), configuration and policy changes via a management interface, and possibly packets from the dat-

aplane. In response, the controllers issue forwarding instructions to switches. All input events are asynchronous, and individual controllers may fail at any time. The controller instances may communicate with each other over the dataplane network, or use a separate dedicated network. In either case, controllers may become partitioned.

The goal of a network control plane is to configure the switch forwarding entries so as to enforce one or more invariants, such as connectivity (*i.e.*, ensuring that a route exists between every endpoint pair), isolation and access control (*i.e.*, various limitations on connectivity), and virtualization (*i.e.*, ensuring that packets are handled in a manner consistent with what would happen in the specified virtual network). A bug causes an invariant to be violated. Invariants can be violated because the system was improperly configured (*e.g.*, the management system [93] or a human improperly specified their goals to the SDN control plane), or because there is a bug within the SDN control plane itself. In this chapter we focus on troubleshooting bugs in the SDN control plane after it has been given a configuration.[2]

In commercial SDN development, software developers work with a team of QA engineers whose job is to find bugs. The QA engineers exercise automated test scenarios that involve sequences of external (input) events such as failures or policy changes on large (software emulated or hardware) network testbeds. If they detect an invariant violation, they hand the resulting trace to a developer for analysis.

The space of possible bugs is enormous, and it is difficult and time consuming to link the symptom of a bug (*e.g.* a routing loop) to the sequence of events in the QA trace (which includes both the external events and internal monitoring data), since QA traces contain a wealth of extraneous events. Consider that an hour long QA test emulating event rates observed in production could contain 8.5 network error events per minute [45] and 500 VM migrations per hour [107], for a total of $8.5 \cdot 60 + 500 \approx 1000$ inputs.

## 3.3 Problem Definition

We represent the forwarding state of the network at a particular time as a configuration $c$, which contains all the forwarding entries in the network as well as the liveness of the various network elements. The control software is a system consisting of one or more controller processes that takes a sequence of external network events $E = e_1 \rightarrow e_2 \rightarrow \cdots e_m$ (*e.g.* link failures) as inputs, and produces a sequence of network configurations $C = c_1, c_2, \ldots, c_n$.

[2] This does not preclude STS from troubleshooting misspecified policies (misconfigurations) so long as test invariants [62] are specified separately.

An invariant is a predicate $P$ over forwarding state (a safety condition, *e.g.* loop-freedom). We say that configuration $c$ violates the invariant if $P(c)$ is false, denoted $\overline{P}(c)$.

We are given an execution trace $L$ generated by a centralized QA test orchestrator.[3] The execution trace $L$ contains a sequence of events

$$\tau_L = e_1 \rightarrow i_1 \rightarrow i_2 \rightarrow e_2 \rightarrow \cdots e_m \rightarrow \cdots i_p$$

which includes external events $E_L = e_1, e_2 \cdots e_m$ injected by the orchestrator, and internal events $I_L = i_1, i_2 \cdots i_p$ triggered by the control software (*e.g.* OpenFlow messages). The external events include timestamps $\{(e_k, t_k)\}$ recorded from the orchestrator's clock.

A replay of execution trace $L$ involves replaying the external events $E_L$, possibly taking into account the occurrence of internal events $I_L$. We denote a replay attempt by $replay(\tau)$. The output of $replay$ is a sequence of forwarding state configurations $C_R = \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n$. Ideally $replay(\tau_L)$ produces the same sequence of network configurations that occurred originally, but as we discuss later this does not always hold.

If the configuration sequence $C_L = c_1, c_2, \ldots, c_n$ associated with the execution trace $L$ violated predicate $P$ (*i.e.*$\exists_{c_i \in C_L}.\overline{P}(c_i)$) then we say $replay(\tau) = C_R$ reproduces that violation if $C_R$ contains another faulty configuration (*i.e.*$\exists_{\hat{c}_i \in C_R}.\overline{P}(\hat{c}_i)$).

The goal of our work is, when given an execution trace $L$ that exhibited an invariant violation,[3] to find a small, replayable sequence of events that reproduces that invariant violation. Within the context of this chapter, we define a minimal causal sequence (MCS) to be a sequence $\tau_M$ where the external events $E_M \in \tau_M$ are a subsequence of $E_L$ such that $replay(\tau_M)$ reproduces the invariant violation, but for all proper subsequences $E_N$ of $E_M$ there is no sequence $\tau_N$ such that $replay(\tau_N)$ reproduces the violation. Note that an MCS is not necessarily *globally* minimal, in that there could be smaller subsequences of $E_L$ that reproduce this violation, but are not a subsequence of this MCS.

We find approximate MCSes by deciding which events to eliminate and, more importantly, when to inject external events. The key component of this system is a mock network that can execute $replay()$. Our focus is on using the mock network to generate random inputs (shown in Table 3.2), detecting bugs in control software, and then finding MCSes that trigger them. We describe this process in the next section.

## 3.4 Approach

Given an execution trace $L$ generated from testing infrastructure,[3] our goal is to find an approximate MCS, so that a human can examine the reduced execution rather than the full trace. This involves two tasks: searching through subsequences of $E_L$, and decid-

[3] We discuss how these execution traces are generated in chapter 3.5.

ing when to inject external events for each subsequence so that, whenever possible, the invariant violation is retriggered.

### 3.4.1 Searching for Subsequences

Checking random subsequences of $E_L$ would be one viable but inefficient approach to achieving our first task. We do better by employing the delta debugging algorithm [135], a divide-and-conquer algorithm for isolating fault-inducing inputs (described in chapter 2). In our case, we use delta debugging to iteratively select subsequences of $E_L$ and replay each subsequence with some timing $T$. If the bug persists for a given subsequence, delta debugging ignores the other inputs, and proceeds with the search for an MCS within this subsequence. See chapter 2.1 for a detailed explanation of the delta debugging algorithm.

The input subsequences chosen by delta debugging are not always valid. Of the possible inputs sequences we generate (shown in Table 3.2), it is not sensible to replay a recovery event without a preceding failure event, nor to replay a host migration event without modifying its starting position when a preceding host migration event has been pruned. Our implementation of delta debugging therefore prunes failure/recovery event pairs as a single unit, and updates initial host locations whenever host migration events are pruned so that hosts do not magically appear at new locations.[4] These two heuristics account for validity of all network events shown in Table 3.2. We do not yet support network policy changes as events, which have more complex semantic dependencies.[5]

### 3.4.2 Searching for Timings

Simply exploring subsequences $E_S$ of $E_L$ is insufficient for finding MCSes: the timing of when we inject the external events during replay is crucial for reproducing violations.

**Existing Approaches.** The most natural approach to scheduling external events is to maintain the original wall-clock timing intervals between them. If this is able to find all reduction opportunities, *i.e.* reproduce the violation for all subsequences that are a superset of some MCS, we say that the inputs are isolated. The original applications of delta debugging [135] make this assumption (where a single input is fed to a single program), as well as QuickCheck's input "shrinking" [24] when applied to blackbox systems like the synchronous part of telecommunications protocols [8].

We tried this approach for reducing our executions, but were rarely able to reproduce invariant violations. As our case studies demonstrate (chapter 3.6), this is largely due to

[4]Handling invalid inputs is crucial for ensuring that the delta debugging algorithm finds a minimal causal subsequence. See chapter 2.1 for a detailed discussion.

[5]If codifying the semantic dependencies of policy changes turns out to be difficult, one could just employ the more expensive version of delta debugging to account for inconsistency [136].

the concurrent, asynchronous nature of distributed systems; consider that the network can reorder or delay messages, or that controllers may process multiple inputs simultaneously. Inputs injected according to wall-clock time are not guaranteed to coincide correctly with the current state of the control software.

We must therefore consider the internal events of the control software. To deterministically reproduce bugs, we would need visibility into every I/O request and response (*e.g.* clock values or network messages), as well as all thread scheduling decisions for each controller. This information is the starting point for thread schedule reduction techniques, which seek to reduce thread interleavings leading up to race conditions. These approaches involve iteratively feeding a single input (the thread schedule) to a single entity (a deterministic scheduler) [22, 25, 57], or statically analyzing feasible thread schedules [52].

A crucial constraint of these approaches is that they must keep the inputs fixed; that is, the controller behavior must depend uniquely on the thread schedule. Otherwise, the controllers may take a divergent code path. If this occurs some processes might issue a previously unobserved I/O request, and the replayer will not have a recorded response; worse yet, a divergent process might deschedule itself at a different point than it did originally, so that the remainder of the recorded thread schedule is unusable to the replayer (since the original preemption points may no longer be executed).

By fixing the inputs, these approaches are forced to stay on the original code path, and are unable to find alternate paths that still trigger the invariant violation. They can only indirectly reduce inputs by truncating thread executions (*i.e.* causing them to exit early), or by removing threads that are entirely extraneous. They consequently strive for a subtly different goal than ours: reducing thread context switches rather than input events.

With additional information obtained by control flow and data flow analysis [71, 109, 53] however, the inputs no longer need to be fixed. The internal events considered by these techniques are individual instructions executed by the programs (obtained by instrumenting the language runtime), in addition to I/O responses and the thread schedule. With access to the instruction-level execution, they can compute program flow dependencies, and thereby remove input events from anywhere in the trace as long as they can prove that doing so cannot possibly cause the faulty execution path to diverge.

Although reduction augmented with control flow and data flow analysis is able to reduce inputs rather than thread context switches, these techniques still do not find alternate executions that trigger the same invariant violation. They are also overly conservative in removing inputs (*e.g.* EFF takes the transitive closure of all possible dependencies [71]) causing them to miss opportunities to remove dependencies that actually semantically commute.

**Allowing Divergence.** Our approach is to dynamically respond to I/O requests during execution reduction rather than recording all I/O requests and thread scheduling decisions. This has several advantages. Unlike the other approaches, we can find shorter alternate code paths that still trigger the invariant violation, since we are not constrained to executing the exact code path from the original run. Previous *best-effort* execution reduction techniques [26, 115] also allow alternate code paths, but do not systematically consider concurrency and asynchrony.[6] We also avoid the runtime overhead of recording all I/O requests and later replaying them (*e.g.* EFF incurs roughly 10x slowdown during replay due to the overhead of code tracing [71]). Lastly, we avoid the extensive effort required to instrument the control software's language runtime, needed by the other approaches to implement a deterministic thread scheduler, interpose on syscalls, or perform program flow analysis. By avoiding assumptions about the language of the control software, we were able to easily apply our system to five different control platforms written in three different languages.[7]

**Accounting for Interleavings.** To reproduce the invariant violation (whenever $E_S$ is a superset of an MCS) we try to inject each input event $e$ only after all other events, including internal events triggered by the control software itself, that precede it in the happens-before relation [68] from the original execution ($\{i \mid i \rightarrow e\}$) have occurred [110]. The intuition behind this heuristic is that we know that the original execution triggered the invariant violation, so we should strive to stay as close to the causal (happens-before) structure of the original execution as possible (while we prune subsequences of the external events).

The internal events we consider are (a) message delivery events, either between controllers (*e.g.* database synchronization messages) or between controllers and switches (*e.g.* OpenFlow commands), and (b) state transitions within controllers (*e.g.* a backup node deciding to become master). Our test orchestrator obtains visibility into (a) by interposing on all messages within the test environment (to be described in chapter 3.5). It optionally obtains partial visibility into (b) by instrumenting controller software with a simple interposition layer (to be described in chapter 3.5.2). By virtue of controlling inputs and message deliveries from a central location, we are able to totally-order the event trace $\tau_L$.

Note that (in this chapter) we do not have control over the occurrence of internal events, so we do not attempt to reduce them. Crucially though, we want to ensure that the ordering of input and internal events during $replay()$ of each subsequence is as close as possible to the original happens-before relation, so that we can find invariant violations (reduction opportunities) and report them to the delta debugging procedure as often as

[6] Park et al.[96] also reproduce multithreaded executions in a best-effort fashion (allowing for alternate code paths), but do not reduce the execution or consider event modifications.

[7] Some of the controllers actually comprise multiple languages.

| Internal message | Masked values |
|---|---|
| OpenFlow messages | xac id, cookie, buffer id, stats |
| packet_out/in payload | all values except src, dst, data |
| Log statements | varargs parameters to printf |

Table 3.1: Internal messages and their masked values.

possible. Our test orchestrator therefore uses its interposition on internal messages to reorder or delay as necessary during replay.

Maintaining the happens-before relation from the original trace (which reproduces the violation) throughout replay of subsequences of the trace (which may or may not reproduce that violation) requires us to address three issues: coping with syntactic differences in internal events across runs, handling internal events from the original execution that may not occur after pruning, and dealing with new internal events that were not observed at all in the original execution.

**Functional Equivalence.** Internal events may differ syntactically when replaying a subsequence of the original execution trace. For example, consider sequence numbers, which are incremented by one for every message sent or received. When replaying a subsequence of the external events, the sequence numbers of internal messages may all differ in value from the original execution.

We observe that many internal events are *functionally equivalent*, in the sense that they have the same effect on the state of the system with respect to triggering the invariant violation (despite syntactic differences). For example, `flow_mod` messages may cause switches to make the same change to their forwarding behavior even if their transaction ids differ.

We leverage this observation by defining masks over semantically extraneous fields of internal events.[8] We only need to define these masks once (in our case, we only define them once for the OpenFlow protocol), and we can then programmatically apply our defined masks during execution reduction from there on out.

We show the OpenFlow header fields we mask in Table 3.1. We consider an internal event $i'$ observed in the replay equivalent (in the sense of inheriting all of its happens-before relations) to an internal event $i$ from the original execution trace if and only if all unmasked fields have the same value and $i$ occurs between $i'$'s preceding and succeeding inputs in the happens-before relation.

**Handling Absent Internal Events.** Some internal events from the original execution trace

[8]One consequence of applying masks is that bugs involving masked fields are outside the purview of our approach.

| Input Type | Implementation |
|---|---|
| Switch failure/recovery | TCP teardown |
| Controller failure/recovery | `SIGKILL` |
| Link failure/recovery | `ofp_port_status` |
| Controller partition | `iptables` |
| Dataplane packet injection | Network namespaces |
| Dataplane packet drop | Dataplane interposition |
| Dataplane packet delay | Dataplane interposition |
| Host migration | `ofp_port_status` |
| Control message delay | Controlplane interposition |
| Non-deterministic TCAMs | Modified switches |

Table 3.2: Input types currently supported by STS.

that causally "happen before" some external input may be absent when we replay a subsequence of that execution. For instance, if we prune an (external) link failure event, then the corresponding link failure notification message will never arise.

We handle this possibility by attempting to "infer" the presence of expected internal events before we replay each external event subsequence. Our algorithm (called PEEK()) for inferring the presence of internal events is depicted in Figure 3.1. The algorithm injects each input, records a checkpoint[9] of the network and the control software's state, allows the system to proceed up until the following input (plus a small time $\epsilon$), records the observed events, and matches the recorded events with the functionally equivalent internal events observed in the original trace.[10]

**Handling New Internal Events.** The last possible change induced by pruning is the occurrence of new internal events that were not observed in the original execution trace. New internal events indicate that there are multiple possibilities for where we might inject the next input event. Consider the following case: if $i_2$ and $i_3$ are internal events observed during replay that are both in the same equivalence class as a single event $i_1$ from the original run, we could inject the next input after $i_2$ or after $i_3$.

In the general case it is always possible to construct two state machines (programs under test) that lead to differing outcomes: one that only leads to the invariant violation when we inject the next input *before* a new internal event, and another only when we inject *after* a new internal event. In other words, to be guaranteed to traverse any existing suffix through the program under test's state machine that leads to the invariant violation, we must recursively branch, trying both possibilities for every new internal event. This

[9] We discuss the implementation details of checkpointing in 3.5.3.

[10] In the case that, due to non-determinism, an internal event occurs during PEEK() but does not occur during replay, we time out on internal events after $\epsilon$ seconds of their expected occurrence.

**Algorithm 3.4.1:** PEEK($events$)

**procedure** PEEK($input\ subsequence$)
$inferred \leftarrow [\,]$
**for** $e_i$ *in subsequence*
$checkpoint\ system$
$inject\ e_i$
$\Delta \leftarrow |e_{i+1}.time - e_i.time| + \epsilon$
$record\ events\ for\ \Delta\ seconds$
$matched \leftarrow original\ events\ \&\ recorded\ events$
$inferred \leftarrow inferred + [e_i] + matched$
$restore\ checkpoint$
**return** $(inferred)$

Figure 3.1: PEEK determines which internal events from the original sequence occur for a given subsequence.

implies an exponential number of possibilities to be explored in the worst case (we discuss this intractability more formally in chapter 4).

Exponential search over these possibilities is not a practical option. In this chapter, our heuristic is to simply ignore new internal events (keeping them pending in the network and never delivering them), always injecting the next input when its last expected predecessor either occurs or times out. This ensures that we always find suffixes through the state machine that contain a subset of the (equivalent) original internal events, but leaves open the possibility of finding divergent suffixes that lead to the invariant violation.

**Recap.** We combine the above heuristics to replay the execution for each external event subsequence chosen by delta debugging: we compute functional equivalency for each internal event intercepted by our interposition layer, we invoke PEEK() to infer absent internal events, and with these inferred causal dependencies we replay the subsequence, waiting to inject each input until each of its (functionally equivalent) predecessors have occurred while allowing unexpected messages through immediately.

### 3.4.3 Complexity

The delta debugging algorithm terminates after $\Omega(\log n)$ invocations of $replay$ in the best case, and $O(n)$ in the worst case, where $n$ is the number of inputs in the original trace [135]. Each invocation of $replay$ takes $O(n)$ time (one iteration for PEEK() and one it-

eration for the replay itself), for an overall runtime of $\Omega(n \log n)$ best case and $O(n^2)$ worst case replayed inputs.

The runtime can be decreased by parallelizing delta debugging: speculatively replaying subsequences in parallel, and joining the results. Storing periodic checkpoints of the system's state throughout testing can also reduce runtime, as it allows us to replay starting from a recent checkpoint rather than the beginning of the trace.

## 3.5 Systems Challenges

Thus far we have assumed that we are given a faulty execution trace, generated by a network testbed. We now provide an overview of how we use a testbed to obtain execution traces, and then describe our system for reducing them.

**Obtaining Traces.** All three of the commercial SDN companies that we know of employ a team of QA engineers to fuzz test their control software on network testbeds, as depicted in Figure 3.2. This fuzz testing infrastructure consists of the control software under test, the network testbed (which may be software or hardware), and a centralized test orchestrator that chooses input sequences, drives the behavior of the testbed, periodically checks invariants, and manages log files. When a bug is discovered, a QA engineer triages it and then sends logs to a software developer for further troubleshooting.

We do not have access to such a QA testbed, and instead built our own. Our testbed mocks out the control plane behavior of network devices in lightweight software switches and hosts (with support for minimal data plane forwarding). We then run the control software on top of this mock network and connect the software switches to the controllers. The mock network manages the execution of events from a single location, which allows it to record a serial event ordering. This design is similar to production software QA testbeds, and is depicted in Figure 3.3. One distinguishing feature of our design is that the mock network interposes on all communication channels, allowing it to delay, drop, or reorder messages as needed to induce failure modes that might be seen in real, asynchronous networks.

We use our mock network to perform testing on controllers to find bugs. Most commonly we generate random input sequences based on event probabilities that we assign (*cf.* chapter 3.6.8), and periodically check the network for invariant violations.[11] We also sometimes run the mock network interactively, where we manually direct each event that mock network injects. Interactive mode is useful for examining the state of the network and developing an intuition for how to to induce event orderings that we believe may

[11] We currently support the following invariants: (a) all-to-all reachability, (b) loop freeness, (c) blackhole freeness, (d) controller liveness, and (e) POX ACL compliance.

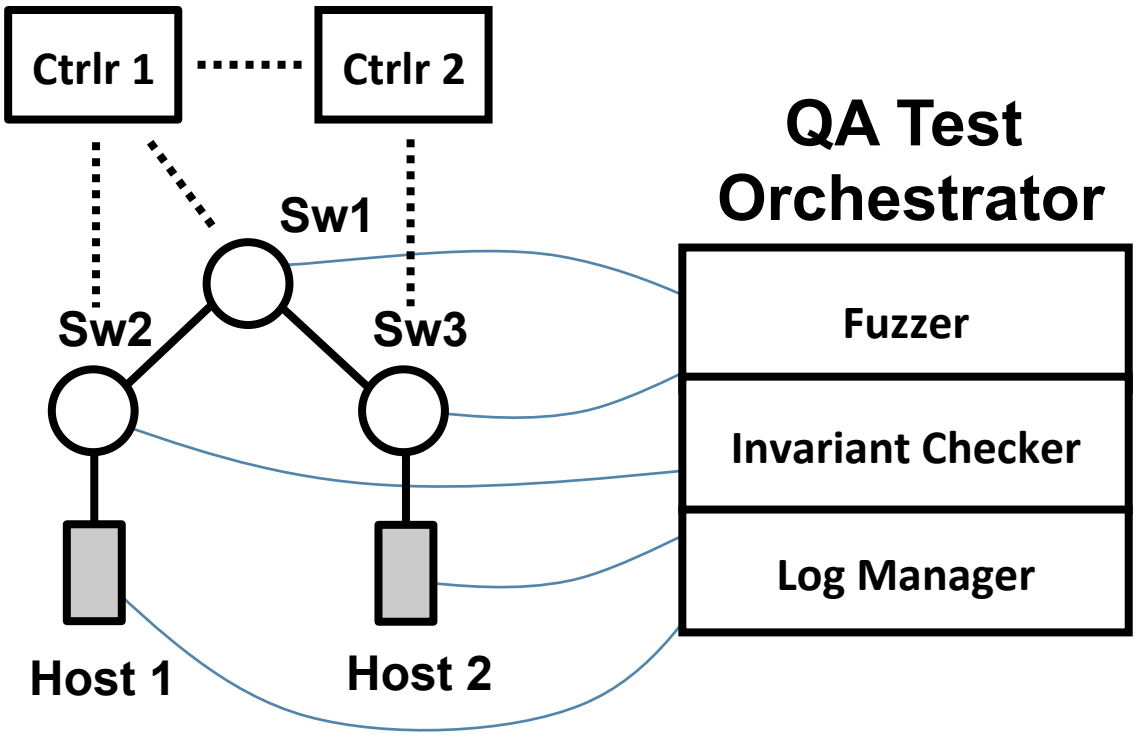



Figure 3.2: Typical QA testbed. A centralized test orchestrator injects inputs and checks invariants

trigger bugs.

**Reducing Executions.** After discovering an invariant violation of interest, we apply delta debugging to reduce the recorded trace. We use the testing infrastructure itself to $replay()$ the events for each of delta debugging's external event subsequences. During $replay()$ the mock network enforces event orderings according to the heuristics we developed in chapter 3.4. It enforces event orderings by buffering any message it receives through interposition, and carefully scheduling the messages that it allows to be delivered. To account for internal events that the mock network does not interpose on (e.g., internal events within a controller that do not involve message sends), the mock network waits a certain period of time between injecting each external event.

Consider the following example of how the mock network performs event scheduling: if the original trace included a link failure preceded by the arrival of a heartbeat message from the controller, during $replay()$ the mock network waits until it observes a functionally equivalent ping probe to arrive, allows the probe through, then tells the switch to fail its link.

STS is our implementation of this system. STS is implemented in more than 23,000 lines of Python in addition to the Hassel network invariant checking library [62]. STS also

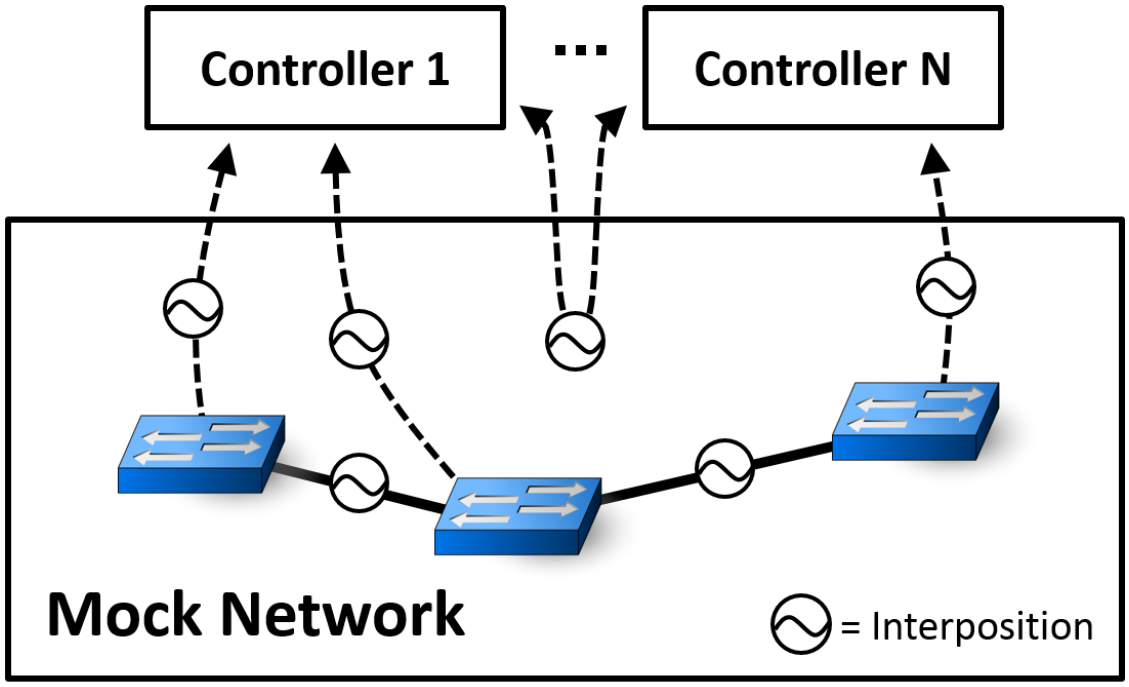



Figure 3.3: STS runs mock network devices, and interposes on all communication channels.

optionally makes use of Open vSwitch [97] as an interposition point for messages sent between controllers. We have made the code for STS publicly available at ucb-sts.github.com/sts.

**Integration With Existing Testbeds.** In designing STS we aimed to make it possible for engineering organizations to implement the technology within their existing QA test infrastructure. Organizations can add delta debugging to their test orchestrator, and optionally add interposition points throughout the testbed to control event ordering during replay. In this way they can continue running large scale networks with the switches, middleboxes, hosts, and routing protocols they had already chosen to include in their QA testbed.

We avoid making assumptions about the language or instrumentation of the software under test in order to facilitate integration with preexisting software. Many of the heuristics we describe below are approximations that might be made more precise if we had more visibility and control over the system, *e.g.* if we could deterministically specify the thread schedule of each controller.

### 3.5.1 Coping with Non-Determinism

Non-determinism in the execution of concurrent processes stems from several sources: differences in system call return values, process scheduling decisions (which can even affect the result of individual instructions, such as x86's interruptible block memory instructions [30]), and asynchronous signal delivery. These sources of non-determinism can affect whether STS is able to reproduce the original bug during replay.

Most testing systems, such as the QA testing frameworks we are trying to improve, do not mitigate non-determinism. STS's main approach to coping with non-determinism is to replay each subsequence chosen by delta debugging multiple times. If the non-deterministic bug occurs with probability $p$, we can (assuming that non-determinism is an i.i.d. random variable) model the probability[12] that we will observe it within $r$ replays as $1-(1-p)^r$. This exponential works strongly in our favor; for example, even if the original bug is triggered in only 20% of replays, the probability that we will not trigger it during an intermediate replay is approximately 1% if we replay 20 times per subsequence.[13]

### 3.5.2 Mitigating Non-Determinism

When non-determinism acutely impacts replay, one might seek to prevent non-determinism altogether. However, this often involves significant engineering effort and substantial runtime overheads. In this chapter, we investigate whether we can successfully reduce executions without fully controlling non-determinism.

In lieu of complete control over non-determinism, we place STS in a position to record and replay all network events in serial order, and ensure that all data structures within STS were unaffected by randomness. For example, we avoid using hashmaps that hash keys according to their memory address, and sort all list return values.

In some cases, a small amount of interposition on the controller software can be beneficial. Routing the `gettimeofday()` syscall through STS helps ensure timer accuracy.[14][15] When sending data over multiple sockets, the operating system exhibits non-determinism in the order it schedules I/O operations. STS optionally ensures a deterministic order of messages by multiplexing all sockets onto a single true socket. On the controller side STS currently adds a shim layer atop the control software's socket library,[16] although this

[12] This probability could be improved by guiding the thread schedule towards known error-prone interleavings [95, 96].

[13] See chapter 3.6.5 for an experimental evaluation of this model.

[14] When the pruned trace differs from the original, we make a best-effort guess at what the return values of these calls should be. For example, if the altered execution invokes `gettimeofday()` more times than we recorded in the initial run, we interpolate the timestamps of neighboring events.

[15] Only supported for POX and Floodlight at the moment.

[16] Only supported for POX at the moment.

could be achieved transparently with a libc shim layer [41].

Controllers also go through internal state transitions that do not involve message sends (which STS already interposes on). With a small amount of interposition on the control software's logging library however, STS can gain gain visibility into the most relevant internal state transitions. This interposition[15] works as follows: whenever a control process executes a log statement, which often indicates that an important state transition is about to take place, we notify STS. Such coarse-grained visibility into internal state transitions does not handle all cases, but we find it suffices in practice.[17] We can also optionally use logging interposition as a synchronization barrier, by blocking the process when it executes crucial logging statements until STS explicitly tells the process that it may proceed.

If blocking was enabled during recording, we force the control software to block at internal state transition points again during replay until STS gives explicit acknowledgment.

### 3.5.3 Checkpointing

To efficiently implement the PEEK() algorithm depicted in Figure 3.1 we assume the ability to record checkpoints (snapshots) of the state of the system under test. We currently implement checkpointing for the POX controller[18] by telling it to `fork()` itself and suspend its child, transparently cloning the sockets of the parent (which constitute shared state between the parent and child processes, since the socket state is managed by the kernel), and later resuming the child. This simple mechanism does not work for controllers that use other shared state such as disk. To handle other shared state one could checkpoint processes within lightweight Unix containers [76]. For distributed controllers, one would also need to implement a consistent cut algorithm to account for in-flight messages when taking the snapshot [19], which is available in several open source implementations [6].

If developers do not choose to employ checkpointing, they can use our implementation of PEEK() that replays all inputs from the beginning of the execution, thereby increasing replay runtime by a factor of $n$. Alternatively, they can avoid PEEK() and solely use the event scheduling heuristics described in chapter 3.5.4.

Beyond its use in PEEK(), snapshotting has three advantages. As mentioned in chapter 3.4.3, only considering events starting from a recent checkpoint rather than the beginning of the execution decreases the number of events to be reduced. By shortening the replay time, checkpointing coincidentally helps cope with the effects of non-determinism, as there is less opportunity for divergence in timing. Lastly, checkpointing can improve

[17] We discuss this limitation further in chapter 3.5.7.

[18] We only use the event scheduling heuristics described in chapter 3.5.4 for the other controllers.

the runtime of delta debugging, since many of the subsequences chosen throughout delta debugging's execution share common input prefixes.

### 3.5.4 Timing Heuristics

We have found a number of heuristics useful for ensuring that invariant violations are consistently reproduced during replay. These heuristics may be used alongside or instead of PEEK(). We evaluated the effectiveness of these heuristics using visualization tools (described in chapter 3.5.5) to compare replay executions with and without the heuristics enabled.

**Event Scheduling.** If we had perfect visibility into the internal state transitions of control software, we would be able to systematically explore the space of possible non-deterministic choices. Unfortunately this level of interposition is difficult to obtain for general software systems. Instead, we develop a few heuristics for accounting for internal events that we do not interpose on.

We find that keeping the wall-clock spacing between replay events close to the recorded timing helps (but does not alone suffice) to ensure that invariant violations are consistently reproduced. When replaying events, we `sleep()` between each event for the same duration that was recorded in the original trace, less the time it takes to replay each event. Accounting for the extra time it takes to replay events is especially important when we time out on internal events, or when input events take a long time to inject.

**Whitelisting keepalive messages.** We observed during some of our experiments that the control software incorrectly inferred that links or switches had failed during replay, when it had not done so in the original execution. Upon further examination we found in these cases that LLDP and OpenFlow echo packets periodically sent by the control software were staying in STS's buffers too long during replay, such that the control software would time out on them. To avoid these differences in timing we added an option to always pass through keepalive messages that mitigates the issue. The limitation of this heuristic is that it cannot be used on bugs involving keepalive messages.

**Whitelisting dataplane events.** Dataplane forward/drop events constitute a substantial portion of overall events. However, for many of the controller applications we are interested in, dataplane forwarding is only relevant insofar as it triggers control plane events (*e.g.* host discovery). We find that allowing dataplane forward events through by default, *i.e.* never timing out on them during replay, can greatly decrease skew in wall-clock timing.

**Using logging statements as barriers.** We also experimented with using logging statements within control software to manipulate its execution speed, for use in the rare cases in which we observed high variability in the controllers' response time. Our technique

is to cause our logging interposition layer to block the entire controller process each time it issues a logging statement until STS gives it explicit permission to proceed. We found that some care is needed to deal with unexpected state transitions, since the controller process will block indefinitely until STS gives it acknowledgment. We currently turn this heuristic off by default.

### 3.5.5 Debugging Tools

Throughout our experimentation with STS, we often found that reduced event traces alone were insufficient to pinpoint the root causes of bugs. We therefore implemented a number of complementary debugging tools within STS, which we use along with Unix utilities to help us complete the final stage of debugging. We illustrate their use in chapter 3.6.

**OFRewind**. STS supports an interactive replay mode similar to OFRewind [123] that allows troubleshooters to query the state of the network throughout replay, filter subsets of the events, check additional invariants, and even induce new events that were not part of the original event trace. Similar to OFRewind, we do not run concurrent controller processes while the user is interactively performing replay, since proper replay across concurrent processes requires precise timing. Instead, STS replays the exact OpenFlow commands from the original trace to the switches, and creates mock TCP connections that drop any messages sent to the controllers.

**Packet Tracing**. Especially for SDN controllers that react to flow events, we found it useful to trace the path of individual packets throughout the network. STS includes tracing instrumentation similar to NetSight [50] for this purpose.

**OpenFlow Reduction**. The OpenFlow commands sent by controller software are often somewhat redundant. For example, controllers may override routing entries, allow them to expire, or periodically flush the contents of flow tables and later repopulate them. STS includes a tool for filtering out such redundant messages and displaying only those commands that are directly relevant to triggering invalid network configurations.

**Event Visualization**. Understanding the timing of messages and internal state transitions is a crucial part of troubleshooting distributed systems. STS includes two visualization tools designed to aid with this task. First, we include a tool to visualize space-time diagrams [68] of event traces. Second, we include a tool to visually highlight event ordering differences between multiple event traces, which is especially useful for comparing the behavior of intermediate delta debugging replays in the face of acute non-determinism.

### 3.5.6 Scaling and Parallelization

When reducing very large event traces we found that the garbage collector for our mock network often became overwhelmed (causing the process to slow down substantially) after replaying several subsequences, since each replay could occupy gigabytes of memory with many small objects. After observing this behavior, we modified STS to fork a process (either local or remote) for each subsequence chosen by delta debugging, and gather the results of the replay via RPC; the OS cleans up the forked process, eliminating garbage collection overhead. As an added benefit, this architectural change allows us to support parallelized delta debugging across multiple cores or machines.

### 3.5.7 Limitations

Having detailed the specifics of our approach we now clarify the scope of our technique's use.

**Partial Visibility.** Our event scheduling algorithm assumes that it has visibility into the occurrence of all relevant internal events. For some controllers this may involve substantial instrumentation effort beyond pre-existing log statements, though as we show in our evaluation, most bugs we encountered can be reduced without perfect visibility.

**Non-determinism.** Non-determinism is pervasive in networking. When non-determinism is present STS (i) replays multiple times per subsequence, and (ii) employs software techniques for mitigating non-determinism, but it may nonetheless output a non-minimal causal sequence. In the common case this is still better than what developers had before, since developers generally do not have tools for reproducing non-deterministic bugs. In the worst case STS leaves the developer where they started: an unpruned execution trace.

**Lack of Guarantees.** Due to partial visibility and non-determinism, we do not provide guarantees on MCS minimality.

**Troubleshooting vs. Debugging.** Our technique is a troubleshooting tool, not a debugger; by this we mean that our approach helps identify and localize inputs that trigger erroneous behavior, but it does not directly identify which line(s) of code cause the error.

**Bugs Outside the Control Software.** Our goal is not to find the root cause of individual component failures in the system (*e.g.* misbehaving routers, link failures). Instead, we focus on how the distributed system as a whole reacts to the occurrence of such inputs.

**Interposition Overhead.** Performance overhead from interposing on messages may prevent STS from reducing bugs triggered by high message rates.[19] Similarly, STS's design may prevent it from reducing extremely large traces, as we evaluate in chapter 3.6.

[19] Although this might be mitigated with time warping [47].

| | Bug Name | Topology | Runtime (s) | Input Size | MCS Size | MCS helpful? |
|---|---|---|---|---|---|---|
| Newly Found | Pyretic loop | 3 switch mesh | 266.2 | 36 | 2 | Yes |
| | POX premature PacketIn | 4 switch mesh | 249.1 | 102 | 2 | Yes |
| | POX in-flight blackhole | 2 switch mesh | 641.1 | 46 | 7 | Yes |
| | POX migration blackhole | 4 switch mesh | 1796.0 | 29 | 3 | Yes |
| | NOX discovery loop | 4 switch mesh | 4990.9 | 150 | 18 | Indirectly |
| | Floodlight loop | 3 switch mesh | 27930.6 | 117 | 13 | Yes |
| | ONOS distributed database locking | 2 switch mesh | N/A | 1 | 1 | N/A |
| Known | Floodlight failover bug | 2 switch mesh | - | 202 | 2 | Yes |
| | ONOS master election | 2 switch mesh | 6325.2 | 30 | 3 | Yes |
| | POX load balancer error checking | 3 switch mesh | 2396.7 | 106 | 24 (N+1) | Yes |
| Synthetic | Null pointer on rarely used codepath | 20 switch FatTree | 157.4 | 62 | 2 | Yes |
| | Overlapping flow entries | 2 switch mesh | 115.4 | 27 | 2 | Yes |
| | Delicate timer interleaving | 3 switch mesh | N/A | 39 | 39 | No |
| | Algorithm misimplementation | 3 switch mesh | 525.2 | 40 | 7 | Indirectly |
| | Multithreaded race condition | 10 switch mesh | 36967.5 | 1596 | 2 | Indirectly |
| | Memory leak | 2 switch mesh | 15022.6 | 719 | 30 (M) | Indirectly |
| | Memory corruption | 4 switch mesh | 145.7 | 341 | 2 | Yes |

Table 3.3: Overview of Case Studies.

**Globally vs. Locally Minimal Input Sequences.** Our approach is not guaranteed to find the globally minimal causal sequence from an input trace, since this involves enumerating the powerset of $E_L$ (a $O(2^n)$ operation). The delta debugging algorithm we employ does (assuming deterministic executions) provably find a locally minimal causal sequence [135], meaning that if any input from the sequence is pruned, no invariant violation occurs.

**Correctness vs. Performance.** We are primarily focused on correctness bugs, not performance bugs.

**Bugs Found Through Fuzzing.** We generate bugs primarily through fuzz testing, not by finding them in operational traces. There is a substantial practical hurdle in instrumenting operational systems to produce execution traces that can be injected into our system, as discussed in chapter 3.7.

**Scaling.** Our discussions with companies with large SDN deployments suggest that scaling to the size of the large execution traces they collect will be a substantial challenge. On the other hand, the fact that these execution traces are so large makes the need for finding MCSes even more acute.

## 3.6 Evaluation

We first demonstrate STS's viability in troubleshooting real bugs. We found seven new bugs by fuzz testing five open source SDN control platforms: ONOS [92] (Java), POX [86]

(Python), NOX [46] (C++), Pyretic [40] (Python), and Floodlight [36] (Java), and debugged these with the help of STS. Second, we demonstrate the boundaries of where STS works well and where it does not by finding MCSes for previously known and synthetic bugs that span a range of bug types encountered in practice.

Our ultimate goal is to reduce developer effort spent on troubleshooting bugs. As this is difficult to measure,[20] since developer skills and familiarity with code bases differs widely, we instead quantitatively show how well STS reduces execution traces, and qualitatively relay our experience using MCSes to debug the newly found bugs.

We show a high-level overview of our results in Table 3.3, and illustrate in detail how STS found MCSes in the rest of this section. Interactive visualizations and replayable event traces for all of these case studies are publicly available at ucb-sts.github.com/experiments.

### 3.6.1 New Bugs

**Pyretic Loop.** We discovered a loop when fuzzing Pyretic's hub module, whose purpose is to flood packets along a minimum spanning tree. After reducing the execution (runtime in Figure 3.4a), we found that the triggering event was a link failure at the beginning of the trace followed some time later by the recovery of that link. After roughly 9 hours over two days of examining Pyretic's code (which was unfamiliar to us), we found what we believed to be the problem in its logic for computing minimum spanning trees: it appeared that down links weren't properly being accounted for, such that flow entries were installed along a link even though it was down. When the link recovered, a loop was created, as the flow entries were still in place. The loop seemed to persist until Pyretic periodically flushed all flow entries.

We filed a bug report along with a replayable MCS to the developers of Pyretic. They found after roughly five hours of replaying the trace with STS that Pyretic told switches to flood out all links before the entire network topology had been learned (including the down link). By adding a timer before installing entries to allow for links to be discovered, the developers were able to verify that the loop no longer appeared. A long term fix for this issue is currently being discussed by the developers of Pyretic.

**POX Premature PacketIn.** We discovered this bug accidentally During a particular fuzzing run, the `l2_multi` module failed unexpectedly with a `KeyError`. The initial trace had 102 input events, and STS reduced it to an MCS of 2 input events as shown in Figure 3.4b.

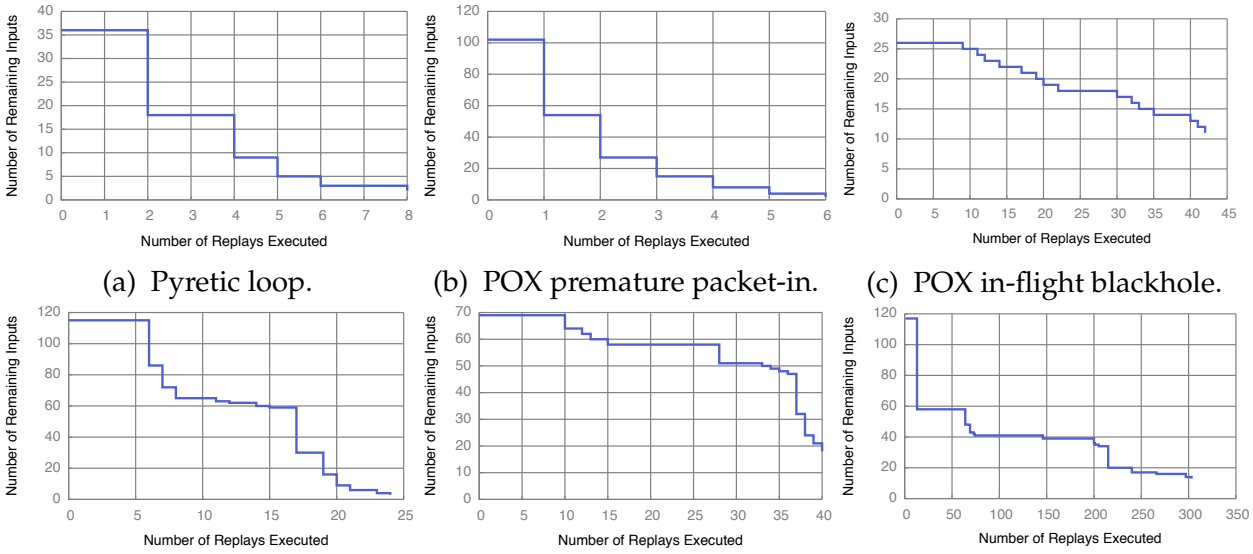
We repeatedly replayed the MCS while adding instrumentation to the POX code. The root cause was a race condition in POX's handshake state machine. The OpenFlow stan-

[20] We discuss this point further in chapter 3.7.

dard requires a 2-message handshake. Afterwards, the switch is free to send arbitrary messages. POX, however, requires an additional series of message exchanges before considering the switch fully connected and notifying the application modules of its presence via a *SwitchUp* event.

In this case, the switch was slow in completing the second part of the handshake, causing the *SwitchUp* to be delayed. During this window, a *PacketIn* (LLDP packet) was forwarded to POX's `discovery` module, which in turned raised a *LinkEvent* to `l2_multi`, which then failed because it expected *SwitchUp* to occur first. We verified with the lead developer of POX that is a true bug.

This case study demonstrates how even a simple handshake state machine can behave unexpectedly and in a non-trivial manner that is hard to understand without being able to repeat the experiment with a reduced trace. Making heavy use of the MCS replay, a developer unfamiliar with the two subsystems was able to root-cause the bug in ˜30 minutes.




(a) Pyretic loop. (b) POX premature packet-in. (c) POX in-flight blackhole.

(d) POX migration blackhole. (e) NOX discovery loop. (f) Floodlight loop.

Figure 3.4: Execution reduction results.

**POX In-flight Blackhole.** We discovered another bug after roughly 20 runs of randomly generated inputs. We noticed a persistent blackhole while POX was bootstrapping its discovery of link and host locations. There were 46 inputs in the initial trace. The initial trace was affected by non-determinism and only replayed successfully 15/20 times. We were able to reliably replay it by employing multiplexed sockets, overriding `gettimeofday()`, and waiting on POX's logging messages. STS returned a 7 input MCS (runtime shown in Figure 3.4c).

We provided the MCS to the lead developer of POX. Primarily using the console output, we were able to trace through the code and identify the problem within 7 minutes, and were able to find a fix for the race condition within 40 minutes. By matching the console output with the code, he found that the crucial triggering events were two in-flight packets (set in motion by prior traffic injection events): POX first incorrectly learned a host location as a result of the first in-flight packet showing up immediately after POX discovered that port belonged to a switch-switch link—apparently the code had not accounted for the possibility of in-flight packets directly following link discovery—and then as a result the second in-flight packet POX failed to return out of a nested conditional that would have otherwise prevented the blackholed routing entries from being installed.

**POX Migration Blackhole.** We noticed after examining POX's code that there might be some corner cases related to host migrations. We set up randomly generated inputs, included host migrations this time, and checked for blackholes. Our initial input size was 115 inputs. STS produced a 3 input MCS (shown in Figure 3.4d): a packet injection from a host ('A'), followed by a packet injection by another host ('B') towards A, followed by a host migration of A. This made it immediately clear what the problem was. After learning the location of A and installing a flow from B to A, the routing entries in the path were never removed after A migrated, causing all traffic from B to A to blackhole until the routing entries expired.

**NOX Discovery Loop.** Next we tested NOX on a four-node mesh, and discovered a routing loop between three switches within roughly 20 runs of randomly generated inputs.

Our initial input size was 68 inputs, and STS returned an 18 input MCS. Our approach to debugging was to reconstruct from the reduced execution trace how NOX should have installed routes, then compare how NOX actually installed routes. This case took us roughly 10 hours to debug. Unfortunately the final MCS did not reproduce the bug on the first few tries, and we suspect this is due to the fact NOX chooses the order to send LLDP messages randomly, and the loop depends crucially on this order. We instead used the console output from the shortest subsequence that did produce the bug (21 inputs, 3 more than the MCS) to debug this trace.

The order in which NOX discovered links was crucial: at the point NOX installed the 3-loop, it had only discovered one link towards the destination. Therefore all other switches routed through the one known neighbor switch. The links adjacent to the neighbor switch formed 2 of the 3 links in the loop.

The destination host only sent one packet, which caused NOX to initially learn its correct location. After NOX flooded the packet though, it became confused about its location. One flooded packet arrived at another switch that was currently not known to be attached to anything, so NOX incorrectly concluded that the host had migrated. Other flooded packets were dropped as a result of link failures in the network and randomly

generated network loss. The loop was then installed when the source injected another packet.

**ONOS distributed database locking.** When testing ONOS, a distributed open-source controller, we noticed that ONOS controllers would occasionally reject switch attempts to connect upon initialization. The initial trace was already reduced, as the initial input was the single event of the switches connecting to the controllers with a particular timing. When examining the logs, we found that the particular timing between the switch connects caused both ONOS controllers to encounter a "failed to obtain lock" error from their distributed graph database. We suspect that the ONOS controllers were attempting to concurrently insert the same key, which causes a known error. We modified ONOS's initialization logic to retry when inserting switches, and found that this eliminated the bug.

**Floodlight loop.** Next we tested Floodlight's routing application. In about 30 minutes, our fuzzing uncovered a 117 input sequence that caused a persistent 3-node forwarding loop. In this case, the controller exhibited significant non-determinism, which initially precluded STS from efficiently reducing the input size. We worked around this by increasing the number of replays per subsequence to 10. With this, STS reduced the sequence to 13 input events in 324 replays and 8.5 hours (runtime shown in Figure 3.4f).

We repeatedly replayed the 13 event MCS while successively adding instrumentation and increasing the log level each run. After about 15 replay attempts, we found that the problem was caused by interference of end-host traffic with ongoing link discovery packets. In our experiment, Floodlight had not discovered an inter-switch link due to dropped LLDP packets, causing an end-host to flap between perceived attachment points.

While this behavior cannot strictly be considered a bug in Floodlight, the case-study nevertheless highlights the benefit of STS over traditional techniques: by repeatedly replaying a significantly reduced execution, we were able to diagnose the root cause—a complex interaction between the LinkDiscovery, Forwarding, and DeviceManager modules.

### 3.6.2 Known bugs

In addition to our troubleshooting case studies, we evaluate STS's ability to reduce traces on a range of bug types, both known and synthetically injected by us.

**Floodlight failover bug.** We were able to reproduce a known problem in Floodlight's distributed controller failover logic [37] with STS. In Floodlight switches maintain one hot connection to a master controller and several cold connections to replica controllers. The *master* holds the authority to modify the configuration of switches, while the other controllers are in *backup* mode and do not change the switch configurations. If a link

fails shortly after the master controller has died, all live controllers are in the backup role and will not take responsibility for updating the switch flow table. At some point when a backup notices the master failure and elevates itself to the master role it will proceed to manage the switch, but without ever clearing the routing entries for the failed link, resulting in a persistent blackhole.

We ran two Floodlight controller instances connected to two switches, and injected 200 extraneous link and switch failures, with the controller crash and switch connect event[21] that triggered the blackhole interleaved among them. We were able to successfully isolate the two-event MCS: the controller crash and the link failure.

**ONOS master election bug.** We reproduced another bug, previously reported in earlier versions and later fixed, in ONOS's master election protocol. If two adjacent switches are connected to two separate controllers, the controllers must decide between themselves who will be responsible for tracking the liveness of the link. They make this decision by electing the controller with the higher ID as the master for that link. When the master dies, and later reboots, it is assigned a new ID. If its new ID is lower than the other controllers', both will incorrectly believe that they are not responsible for tracking the liveness of the link, and the controller with the prior higher ID will incorrectly mark the link as unusable such that no routes will traverse it. This bug depends on initial IDs chosen at random, and ONOS is not instrumented to support deterministic replay of random values. We mitigated this inherent non-determinism by replaying each subsequence 5 times. With this setting, STS was able to reduce the trace to 3 elements.

**POX load balancer error checking.** We are aware that POX applications do not always check error messages sent by switches rejecting invalid packet forwarding commands. We used this to trigger a bug in POX's load balancer application: we created a network where switches had only 25 entries in their flow table, and proceeded to continue injecting TCP flows into the network. The load balancer application proceeded to install entries for each of these flows. Eventually the switches ran out of flow entry space and responded with error messages. As a result, POX began randomly load balancing each subsequent packet for a given flow over the servers, causing session state to be lost. We were able to reduce the execution for this bug to 24 elements (there were two preexisting auxiliary flow entries in each routing table, so 24 additional flows made the 26 (N+1) entries needed to overflow the table). A notable aspect of this MCS is that its size is directly proportional to the flow table space, and developers would find across multiple fuzz runs that the MCS was always 24 elements.

[21] We used a switch connect event rather than a link failure event for logistical reasons, but both can trigger the race condition.

### 3.6.3 Synthetic bugs

Lastly, we injected synthetic bugs across a range of bug types into POX. For space reasons we only briefly describe these bugs.

**Delicate timer interleaving.** We injected a crash on a code path that was highly dependent on the interleaving of internal timers triggered within POX. This is a particularly hard case for STS, since we have little control of internal timers. We were able to trigger the code path during fuzzing, but were unable to reproduce the bug during replay after five attempts, and were left with the original 39 input trace. This is the only case where we were unable to replay trace.

**Algorithm misimplementation.** We modified POX's implementation of Floyd-Warshall to create loops. We noticed that the MCS was inflated by at least two events: a link failure and a link recovery that we did not believe were relevant to triggering the bug we induced. The final MCS also was not replayable on the first try. We suspect that these problems may have been introduced by the fact that the routing implementation depended on the discovery module to find links in the network, and the order in which these links are discovered is non-deterministic.

**Overlapping flow entries.** We ran two modules in POX: a capability manager in charge of providing upstream DoS protection for servers, and a forwarding application. The capabilities manager installed drop rules upstream for servers that requested it, but these rules had lower priority than the default forwarding rules in the switch. We were able to reduce 27 inputs to the two traffic injection inputs necessary to trigger the routing entry overlap.

**Null pointer on rarely used codepath.** On a rarely used-code path, we injected a null pointer exception, and were able to successfully reduce a fuzz trace of 62 events to the expected conditions that triggered that code path: control channel congestion followed by decongestion.

**Multithreaded race condition.** We created a race condition between multiple threads that was triggered by any packet I/O, regardless of input. With 5 replays per subsequence, we were able to reduce a 1596 input in 10 hours 15 minutes to a replayable 2 element failure/recovery pair as an MCS. The MCS itself though may have been somewhat misleading to a developer (as expected), as the race condition was triggered randomly by any I/O, not just these two inputs events.

**Memory leak.** We created a case that would take STS very long to reduce: a memory leak that eventually caused a crash in POX. We artificially set the memory leak to happen quickly after allocating 30 (M) objects created upon switch handshakes, and interspersed 691 other input events throughout switch reconnect events. The final MCS found after 4 hours 15 minutes was exactly 30 events, but it was not replayable. We suspect this was

because STS was timing out on some expected internal events, which caused POX to reject later switch connection attempts.

**Memory corruption.** We simulated a case where the receipt of link failure notification on a particular port causes corruption to one of POX's internal data structures. This corruption then causes a crash much later when the data structure is accessed during the during corresponding port up. These bugs are often hard to debug, because considerable time can pass between the event corrupting the data structure and the event triggering the crash, making manual log inspection or source level debugging ineffective. STS proved effective in this case, reducing a larger trace to exactly the 2 events responsible for the crash.

### 3.6.4 Overall Results & Discussion

We show our overall results in Table 3.3. We note that with the exception of 'Delicate timer interleaving', STS was able to significantly reduce the size of the input traces. As described in the case studies, we were able to counter some sources of non-determinism by replaying multiple times per subsequence and adding instrumentation to controllers.

The cases where STS was most useful were those where a developer would have started from the end of the trace and worked backwards, but the actual root cause lies many events in the past (*e.g.* the Memory corruption example). This requires many reiterations through the code and logs using standard debugging tools (*e.g.* source level debuggers), and is highly tedious on human timescales. In contrast, it was easy to step through a small event trace and manually identify the code paths responsible for a failure.

Bugs that depend on fine-grained thread-interleaving or timers inside of the controller are the worst-case for STS. This is not surprising, as they do not directly depend on the input events from the network, and we do not directly control the internal scheduling and timing of the controllers. The fact that STS has a difficult time reducing these traces is itself indication to the developer that fine-grained non-determinism is at play.

### 3.6.5 Coping with Non-determinism

Recall that STS optionally replays each subsequence multiple times throughout delta debugging to mitigate the effects of non-determinism. We evaluate the effectiveness of this approach on the reduction of a synthetic non-deterministic loop created by Floodlight. Table 3.4 demonstrates that the size of the resulting MCS decreases with the number of replays per subsequence. This suggests that replaying each subsequence multiple times is effective in coping with non-determinism, at the cost of increased runtime.

| Max replays per subsequence | Size of final MCS | Total hours |
|---|---|---|
| 1 | 65 | 6.10 |
| 2 | 20 | 6.37 |
| 3 | 15 | 7.78 |
| 4 | 12 | 9.59 |
| 5 | 9 | 6.38 |
| 6 | 9 | 11.20 |
| 7 | 9 | 11.83 |
| 8 | 6 | 12.35 |
| 9 | 6 | 11.13 |
| 10 | 6 | 12.86 |

Table 3.4: Effectiveness of replaying subsequences multiple times in mitigating non-determinism.

### 3.6.6 Instrumentation Complexity

For POX and Floodlight, we added shim layers to the controller software to redirect `gettimeofday()`, interpose on logging statements, and demultiplex sockets. For Floodlight we needed 722 lines of Java to obtain this indirection, and for POX we needed 415 lines of Python.

### 3.6.7 Scalability

Mocking the network in a single process potentially prevents STS from triggering bugs that only appear at large scale. We ran STS on large FatTree networks to see where these scaling limits exist. On a machine with 6GB of memory, we ran POX as the controller, and measured the time to create successively larger FatTree topologies, complete the OpenFlow handshakes for each switch, cut $5\%$ of links, and process POX's response to the link failures. As shown in Figure 3.5, STS's processing time scales roughly linearly up to 2464 switches (a 45-pod FatTree). At that point, the machine started thrashing, but this limitation could easily be removed by running on a machine with $>$6GB of memory.

Note that STS is not designed for simulating high-throughput dataplane traffic; we only forward what is necessary to exercise the controller software. In proactive SDN setups, dataplane events are not relevant for the control software, except perhaps for host discovery.

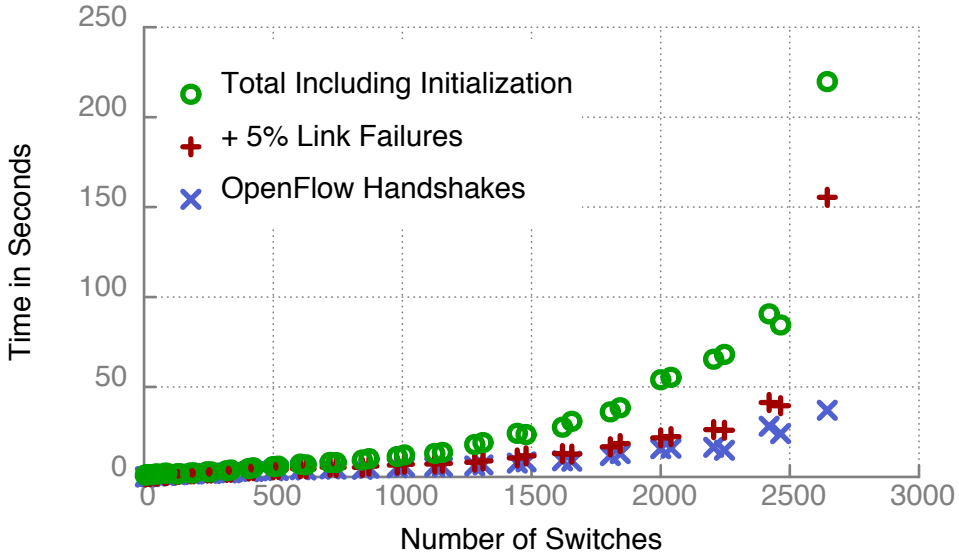



Figure 3.5: Simulation time for bootstrapping FatTree networks, cutting 5% of links, and processing the controller's response.

### 3.6.8 Parameters

Throughout our experimentation we found that STS leaves open several parameters that need to be set properly in order to effectively find and troubleshoot bugs.

**Setting fuzzing parameters.** STS's fuzzer allows the user to set the rates different event types are triggered at. In our experiments with STS we found several times that we needed to set these parameters such that we avoided bugs that were not of interest to developers. For example, in one case we discovered that a high dataplane packet drop rate dropped too many LLDP packets, preventing the controller from successfully discovering the topology. Setting fuzzing parameters remains an important part of experiment setup.

**Differentiating persistent and transient violations.** In networks there is a fundamental delay between the initial occurrence of an event and the time when other nodes are notified of the event. This delay implies that invariant violations such as loops or blackholes can appear before the controller(s) have time to correct the network configuration. In many cases such transient invariant violations are not of interest to developers. We therefore provide a threshold parameter in STS for how long a invariant violation should persist before STS reports it as a problem. In general, setting this threshold depends on the network and the invariants of interest.

**Setting $\varepsilon$.** Our algorithm leaves an open question as to what value $\varepsilon$ should be set to. We experimentally varied $\varepsilon$ on the POX in-flight blackhole bug. We found for both cases that

the number of events we timed out on while isolating the MCS became stable for values above 25 milliseconds. For smaller values, the number of timed out events increased rapidly. We currently set $\varepsilon$ to 100 milliseconds.

In general, larger values of $\varepsilon$ are preferable to smaller values (disregarding runtime considerations), since we can always detect when we have waited too long (*viz.* when a successor of the next input has occurred), but we cannot detect when we have timed out early on an internal event that is in fact going to occur shortly after.

## 3.7 Discussion

**How much effort do MCSes really save?** Based on conversations with engineers and our own industrial experience, two facts seem to hold. First, companies dedicate a substantial portion of their best engineers' time on troubleshooting bugs. Second, the larger the trace, the more effort is spent on debugging, since humans can only keep a small number of facts in working memory [88]. As one developer puts it, "Automatically shrinking test cases to the minimal case is immensely helpful" [11].

**Will this approach work on all controllers?** We make limited assumptions about the controller software in use. Three of the five platforms we investigated were exercised with STS without any modifications. Limited changes to the controller platforms (*e.g.* overriding `gettimeofday()`) can increase replay accuracy further. In general, we expect STS to support controllers conforming to OpenFlow 1.0.

**Why do you focus on SDN?** SDN represents both an opportunity and a challenge. In terms of a challenge, SDN control software—both proprietary and open source—is in its infancy, which means that bugs are pervasive.

In terms of an opportunity, SDN's architecture facilitates the implementation of systems like STS. The interfaces between components of the system (*e.g.* OpenFlow for switches [87] and OpenStack Neutron for management [93]) are well-defined, which is crucial for codifying functional equivalencies. Moreover, the control flow of SDN control software repeatedly returns to a quiescent state after processing inputs, which means that many inputs can be pruned.

In chapter 4, we show the extent to which the ideas embodied in STS are applicable to other types of distributed systems.

**Enabling analysis of production executions.** STS does not currently support reduction of production (as opposed to QA) executions. Production systems would need to include Lamport clocks on each message [68] or have sufficiently accurate clock synchronization to obtain a partial (happens-before) ordering of the message deliveries in the execution. Inputs would also need to need to be logged in sufficient detail for STS to replay a syn-

thetic version. Finally, without care, a single input event may appear multiple times in the distributed logs. The most robust way to avoid redundant input events would be to employ perfect failure detectors [18], which log a failure iff the failure actually occurred.

## 3.8 Conclusion

SDN's purpose is to make networks easier to manage. SDN does this, however, by pushing complexity into SDN control software itself. Just as sophisticated compilers are hard to write, but make programming easy, SDN control software makes network management easier, but only by forcing the developers of SDN control software to confront the challenges of concurrency, asynchrony, partial failure, and other notoriously hard problems inherent to all distributed systems.

In this chapter we demonstrated a technique for automatically reducing the inputs responsible for triggering a given bug, without making assumptions about the language or instrumentation of the software. We focused specifically on SDN control software here, but in the next chapter we will show that these execution reduction heuristics are applicable to other distributed systems.

# Chapter 4

# Reducing Faulty Executions of Distributed Systems, With Application-Layer Interposition

## 4.1 Introduction

In this chapter, we seek to generalize our investigation from chapter 3 along several dimensions. The STS system described in chapter 3 targeted a specific distributed system (SDN controllers), and focused on reducing input events given limited control over the execution. Here we target a broader range of systems, define the general problem of execution reduction, exercise significantly greater control over the execution, and systematically explore the state space. We also articulate new reduction strategies that quickly reduce input events, internal events, and message contents.

The key enabler that allows us to gain control over the execution is a computational model—the actor model—that encapsulates all sources of non-determinism in the network in a small number of interposition points. This makes it easy for us to interpose on practical systems.

With this control in place, it is theoretically possible to simply enumerate all possible schedules, execute each one, and pick the smallest execution that triggers the bug. The space of all possible schedules is intractably large however. Our general approach is to carefully prioritize the order in which we explore the schedule space, with the goal of quickly finding small executions that still trigger the bug. We design these prioritization functions based on our understanding of how programs behave in practice. As we learned in chapter 3, our key insight is that if one schedule triggers a bug, schedules that are similar in their causal structure should have a high probability of also triggering the

bug.

Translating this intuition into a prioritization function requires us reason about the similarity or dissimilarity of two distinct executions. We develop a hierarchy of equivalence relations between events, and show that systematically exploring schedules that are close to the original execution yield significant gains in reduction over the previous heuristics described in chapter 3.

Our tool, Distributed Execution Minimizer (DEMi), is implemented in ~14,000 lines of Scala. We have applied DEMi to akka-raft [4], an open source Raft consensus implementation, and Apache Spark [132], 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 1X and 16X (4X median) smaller than the executions produced by the previous state-of-the-art blackbox technique (chapter 3). 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.

## 4.2 Problem Statement

We start by introducing a model of distributed systems as groundwork for defining our goals. As we discuss further in chapter 4.4.2, we believe this model is general enough to capture the behavior of many practical systems.

### 4.2.1 System Model

Following [34], 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 $\tau$ 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.

### 4.2.2 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.

### 4.2.3 Problem Definition

We are given a schedule $\tau$ injected by a test orchestrator,[1] 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 $\overline{P}$. Within the context of this chapter's formalism, we redefine a minimal causal sequence (MCS) to be a subsequence of external events

[1] We explain how we obtain these schedules in chapter 4.4.

$E' \sqsubseteq E$ 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[2] than $\tau$ containing $E' - e$ that produce $\overline{P}$.[3]

We start by reducing 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 reducing internal events after the external events have been reduced. 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 shrink 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 unexpected 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 [43]), and because mature debugging tools already exist for sequential code.

With a reduced execution in hand, the developer begins debugging. Echoing the benefits of sequential test case reduction, 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.

## 4.3 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?*

[2]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 [2], essentially because nodes cannot distinguish between crashes of their peers and indefinite message delays.

[3]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 [136], we only seek a 1-minimal subsequence $E'$ instead of a globally minimal subsequence.

To find MCSes in reasonable time, we split schedule exploration into two parts. We start by using delta debugging [136] (explained in chapter 3.4.1), a reduction 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', explained in chapter 2.2) to prune this schedule space by eliminating equivalent schedules (i.e. schedules that differ only in the ordering of commutative events [35]). 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 reduction ends once the system has successfully explored all subsequences generated by delta debugging. Limiting schedule exploration to a fixed time budget allows reduction to finish in bounded time, albeit at the expense of completeness (i.e., we may not return a perfectly minimal event sequence).

To further reduce execution length, we continue to use the same schedule exploration procedure to reduce internal events once external event reduction has completed. Internal event reduction continues until no more events can be removed, or until the time budget for execution reduction 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 reduce the number of internal events. With this road map in mind, below we describe our execution reduction approach in greater detail.

### 4.3.1 Choosing Subsequences of External Events

We model the task of reducing 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 chapter 4.3.2.1) 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 [135, 136], 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 explained in detail in chapter 3.4.1.

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 reduction 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 [136].

### 4.3.2 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 [44], many events occurring in a schedule are *commutative*, i.e., the system arrives at the same configuration regardless of the order events are applied. As we explained in detail in chapter 2.2, a well-studied algorithm for removing commutative (equivalent) schedules from the schedule space is Dynamic Partial Order Reduction (DPOR). We make use of DPOR to efficiently explore the schedule space. We then implement our scheduling heuristics as modifications to the original DPOR algorithm.

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 [75]. 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.

#### 4.3.2.1 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 reduce a known buggy execution 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 (containing a given external event subsequence $E'$) they find within their time budget. If a scheduling strategy cannot find a reproducing schedule within its time budget, it returns a special marker $\perp$. We design our key strategy (shown in Algorithm 2) 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 re-triggering 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 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). We show an example initial schedule in Figure 4.1.

**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 4.1). 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 [105, 122].

Some types of message fields obviously exhibit data-independence. Consider authentication cookies. Distributed systems commonly use cookies to track which requests are tied to which users. However, the value of the cookie is not in itself relevant to how the distributed system behaves, as long as each user attaches a consistent value to their messages. This means that two executions with exactly the same users and exactly the same messages except for the cookie fields can be considered equivalent.

To leverage data independence, application developers can (optionally) supply us

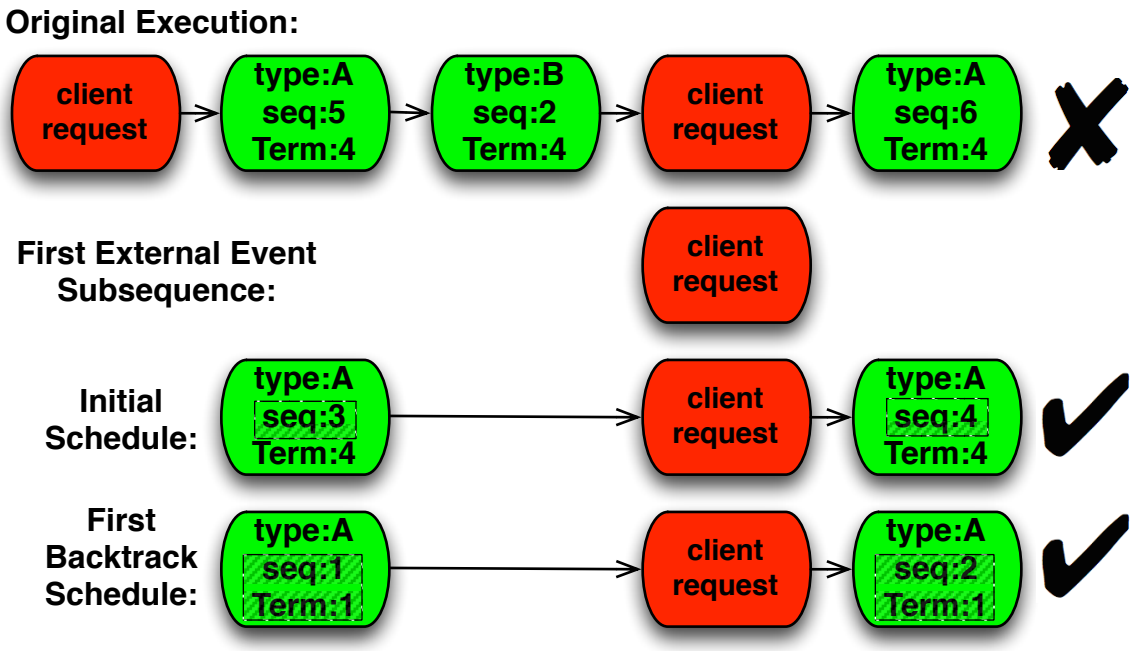



Figure 4.1: Example schedules. External message deliveries are shown in red, internal message deliveries in green. Pending messages, source addresses, and destination addresses are not shown. The 'B' message becomes absent when exploring the first subsequence of external events. We choose an initial 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. The violation is still not triggered, so we continue exploring.

with a 'message fingerprint' function,[4] 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 [105]; 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

[4]It may be possible to extract message fingerprints automatically using program analysis or experimentation [116]. Nonetheless, manually defining fingerprints does not require much effort (see Table 4.5). Without a fingerprint function, we default to matching on message type (Observation #3).

**Algorithm 2** Pseudocode for schedule exploration. TEST is invoked once per external event subsequence $E'$. We elide the details of DPOR for clarity (see chapter 2.2 for a complete description). $\tau$ 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 ($\tau''$[0..i] and $\tau''$[i+1...$\tau''$.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'}
```

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 2. We show an example initial schedule in Figure 4.1.

**Challenge: history-dependent message contents.** This initial schedule can be remarkably effective, as demonstrated by the fact that we 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. Execution reduction 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 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 execution reduction 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 before the message is converted to bits on the wire.

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 4.1, 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 2. Whenever we find a backtrack point (pending message) that matches the type but not the fingerprint of an original delivery event from $\tau$, we replace the original delivery with the backtrack's pending message, and execute the events before and after the backtrack point as before. We also track which schedules we have executed in the past to avoid redundant exploration.

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.

**Reducing 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 reduction continues until there is no more reduction to be performed, or until the time budget for execution reduction 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 reduce 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 reduce 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 reduce internal events, and wherever possible, we seek to shrink external message contents.

### 4.3.3 Comparison to Prior Work

We made observations #1 and #2 in chapter 3. In this chapter, 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 (chapter 3).

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 reduce internal events, and shrink external message contents.

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

Table 4.1: Tasks we assume the application programmer completes in order to test and reduce using DEMi. Defaults of '-' imply that the task is not optional.

## 4.4 Systems Challenges

We implement our techniques in a publicly available tool we call DEMi (Distributed Execution Minimizer) [29]. DEMi is an extension to Akka [3], an actor framework for JVM-based languages. Actor frameworks closely match the system model in chapter 4.2: 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 [65], 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 execution reduction 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 execution reduction 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 Table 4.1), 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),[5] 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 reduce the execution as described in chapter 4.3.

In Table 4.1 we summarize the various tasks, both optional and necessary, that we assume programmers complete in order to test and reduce using DEMi.

**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.

[5] Violation fingerprints should be specific enough to disambiguate different bugs found during execution reduction, 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.

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 execution reduction, 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 chapter 4.2 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 DEMi decides to deliver the response.

### 4.4.1 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 [109]) 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 [126, 35], 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 [30, 41, 74, 134] 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 execution reduction: deterministic replay does not inform how the schedule space should be explored; it only allows one to deterministically replay prefixes of events. Moreover, reducing 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 chapter 5.

**Support for production traces.** DEMi does not currently support reduction 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 reducing 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.

### 4.4.2 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

| 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 4.2: 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 (chapter 3); 'TFB' minus 'Optimal' shows how far from optimal our results are; and 'NoDiverge' shows the size of reduced executions when no divergent schedules are explored (explained in chapter 5).

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.

## 4.5 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 reduction progress within a fixed time budget. We show a high-level overview of our results in Table 4.2. 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" 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 (chapter 3) for each of delta debugging's subsequences. The "TFB" column shows the final execution size after we apply our techniques ('Type Fingerprints with

| Bug Name | STSSched | TFB | Total |
|---|---|---|---|
| raft-45 | 56s (594) | 114s (2854) | 170s (3448) |
| raft-46 | 73s (384) | 209s (4518) | 282s (4902) |
| raft-56 | 54s (524) | 2078s (31149) | 2132s (31763) |
| raft-58a | 137s (624) | 43345s (834972) | 43482s (835596) |
| raft-58b | 23s (340) | 31s (1747) | 69s (2087) |
| raft-42 | 118s (568) | 10558s (176517) | 10676s (177085) |
| raft-66 | 14s (192) | 334s (10334) | 348s (10526) |
| spark-2294 | 330s (248) | 97s (78) | 427s (326) |
| spark-3150 | 219s (174) | 26s (21) | 254s (195) |
| spark-9256 | 96s (73) | 0s (0) | 210s (73) |

Table 4.3: Runtime of execution reduction in seconds (total schedules executed). spark-9256 only had unignorable events remaining after STSSched completed, so TFB was not necessary.

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 reducing external events (our primary reduction target) for most case studies. TFB is significantly more effective for reducing 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 [4] and Apache Spark [132]. akka-raft, as an early-stage software project, demonstrates how DEMi can

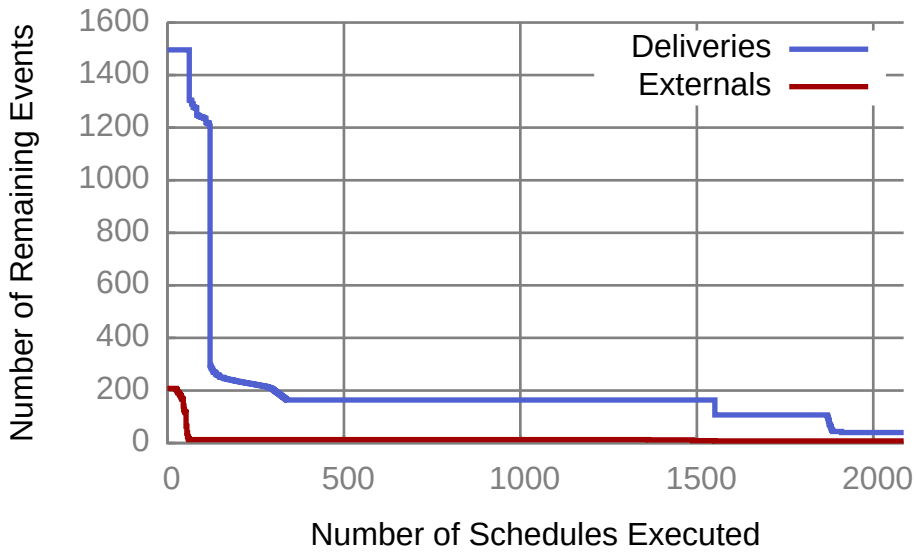



Figure 4.2: Reduction pace for raft-58b. Significant progress is made early on, then progress becomes rare.

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 reduced 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.

**Reduction Pace.** To measure how quickly DEMi makes progress, we graph schedule size as a function of the number of executions DEMi tries. Figure 4.2 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 4.3.[6] The time budget we allotted to DEMi for all 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 execution reduction helps with program-

[6]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 duration for those timers has actually passed, without the application being aware of this fact (cf. [47])

mer productivity. Data on how humans do debugging is scarce; we are aware of only one study that measures how quickly developers debug reduced vs. non-reduced traces[48]. Nonetheless, since humans can only keep a small number of facts in working memory [88], execution reduction seems generally useful. As one developer puts it, "Automatically shrinking test cases to the minimal case is immensely helpful"[11].

### 4.5.1 Raft Case Studies

Our first set of case studies are taken from akka-raft [4]. 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 chapter 4.5.4. 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 unreduced 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 unreduced executions.

### 4.5.2 Spark Case Studies

Spark [7] 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 chapter 4.5.5. 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.

### 4.5.3 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

| | 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 4.4: External message shrinking results for raft-45 starting with 9 processes. Message shrinking + execution reduction 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 4.5: 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.

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 4.4 demonstrate that message shrinking can help reduce both external events and message contents.

**Instrumentation Overhead.** Table 4.5 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 chapter 4.4), specify invariants, and configure DEMi. In total we spent roughly one person-month debugging non-determinism.

### 4.5.4 Full Description of 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 [91]: 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).

For each of the bugs where we did not initially know the root cause, we started debugging by first reduced the failing execution. Then, we walked through the sequence of message deliveries in the reduced 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 4.2 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 reduced 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 reduced 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.

### 4.5.5 Full Description of 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 '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.

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 reduced 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.

## 4.6 Conclusion

In this chapter, we started by observing that the actor model allows engineers to interpose on sources of non-determinism in their distributed system with low effort. Then, we demonstrated that with more complete control over sources of non-determinism, we can achieve significantly greater reduction in execution size (through systematic exploration of the space of possible schedules), as well as a cleaner problem formulation. Within the actor model we articulated new execution reduction strategies that quickly reduce both input events, internal events, and message contents.

The results we found from applying our reduction strategies to two actor-based distributed systems were quite promising, and leave us optimistic that these techniques can be successfully applied to a wide range of distributed and concurrent systems.

# Chapter 5

# Related Work

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

## 5.1 Reduction Techniques

**Input Reduction for Sequential Programs.** Reduction algorithms for sequentially processed inputs are well-studied [136, 24, 99, 121, 16, 48, 20]. These (particularly: delta debugging) form a component of our solution, but they do not consider interleavings of internal events from concurrent processes.

**Reduction without Interposition.** Several tools reduce inputs to concurrent systems without controlling sources of non-determinism [8, 26, 115, 58, 55]. The most sophisticated of these replay each subsequence multiple times and check whether the violation is reproduced at least once [55, 25]. Their major advantage is that they avoid the engineering effort required to interpose. Their drawback, as we found in chapter 3, is that bugs are often not easily reproducible without interposition. Furthermore, without interposition these techniques cannot (directly) reduce internal events.

QuickCheck's PULSE controls the message delivery schedule [25] and supports schedule reduction. During replay, it considers the order messages are sent in, but 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 [54]. As the QuickCheck developers point out, even with PULSE's interposition, QuickCheck's reduced counterexamples can be large [98]. With more complete exploration of the schedule space their reduction would likely be improved.

**Thread Schedule Reduction.** Other techniques seek to reduce thread interleavings leading up to concurrency bugs [22, 57, 52, 31]. These generally work by iteratively feeding a single input (mutations of a recorded thread schedule) to a single entity (a deterministic

thread scheduler). These approaches ensure that the program never diverges from the control flow of the original schedule (otherwise the recorded I/O responses and context switch points from the original execution would become useless). Besides reducing context switches, these approaches at best *truncate* thread executions by having threads exit earlier than they did in the original execution (thereby shortening the execution trace), but they cannot remove extraneous events from the middle of the trace.

Although these thread schedule reduction techniques do not explore divergent schedules,[1] that constraint is not fundamental. Rather than only considering the original recorded thread schedule, one could explore new (divergent) schedules, store the newly recorded I/O responses and context switch points, and replay those recordings deterministically later on. Such a system for multicore schedule reduction would be equivalent to DEMi as we apply it to message passing distributed systems.

**Thread Schedule Reduction Augmented with Program Analysis.** One can remove events in the middle of a recorded deterministic replay log by analyzing the program's control- and dataflow dependencies and providing a proof that removing those events will not shift the execution's context switch points [71, 109, 53, 17, 32, 119]. These techniques do not explore alternate code paths. Program analysis also over-approximates state reachability (because some conditionals are undecidable or difficult to model; e.g., EFF takes the transitive closure of all possible dependencies [71]), disallowing them from removing dependencies that actually commute.

We compare against thread schedule reduction (without divergence) by configuring DEMi to reduce as before, but to abort any execution where it detects a previously unobserved state transition. Column 'NoDiverge' of Table 4.2 shows the results, which demonstrate that divergent executions are crucial to DEMi's reduction gains for the akka-raft case studies.

Although previous program analysis papers did not consider exploring alternate code paths [71, 109, 53, 17, 32, 119], program analysis could be leveraged to help prune DEMi's search space. We plan to investigate this idea in future work.

**Model Checking.** The algorithmic aspects of this dissertation are most closely related to the model checking literature.

Abstract model checkers convert (possibly concurrent) programs to logical formulas, find logical contradictions (assertion violations) using solvers, and reduce the logical conjunctions to aid understanding [23, 63, 84]. 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

[1] PRES explores divergent schedules for best-effort replay of multithreaded executions, but does not reduce executions [96].

libraries.

It is also possible to extract logic formulas from raw binaries [9]. 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, reduction is useful beyond testing, e.g. for simplifying production traces.

Motivated by the intractability of systematic input enumeration, many papers develop heuristics for finding bugs quickly [117, 89, 90, 125, 15, 111, 38, 72, 75, 95, 28]. 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 an application's use of TCP constrains its schedule space.

As far as we know, we are the first to combine DPOR and delta debugging to reduce executions. Others have modified DPOR to keep state [128, 130] and to apply heuristics for choosing initial schedules [70], but these changes are intended to help find new bugs rather than reduce existing faulty executions.

**Program Slicing & Automated Debugging.** Program slicing [120] seeks to find a minimal subset of the statements in a program that could possibly affect the outcome of an assertion. Literature following Weisers' original program slicing paper goes further to try to automatically locate the exact line(s) of code or state transitions that are responsible for a bug, using statistical data [139], test coverage data [56, 124], constraint solving [82], fault injection [67] and experimentally driven executions of the failing program [27, 108]. Our goal is to slice the temporal dimension of an execution rather than the code dimension.

**Log Comprehension.** Model inference techniques summarize log files in order to make them more easily understandable by humans [33, 13, 12, 14, 80, 81]. Model inference is complementary, as it does not modify the event logs.

## 5.2 Debugging for Networked & Concurrent Systems

We end this chapter by discussing literature on the general topic of troubleshooting. For an in-depth overview of systematic troubleshooting techniques, see our HotSDN publication [51].

**Interposition on blackbox distributed systems.** DEMi's software architecture closely resembles other automated testing systems for distributed systems [75, 72, 106]. All of

these tools interpose on timers and message send and receive events. MoDist [75] interposes at the syscall level, whereas SAMC [72] and dBug [106] interpose primarily at the application layer (similar to DEMi). As we discussed in chapter 4.4.2, interposing at the application layer has several advantages over interposing at the syscall level, although syscall level interposition is more broadly applicable.

STS's software architecture closely resembles other network simulators [49, 112, 118]. Most of these simulators are focused on producing high fidelity performance behaviors, whereas we designed STS with the intent of providing precise control over event timings without focusing on performance.

**Root Cause Analysis.** Without perfect instrumentation, it is often not possible to know exactly what events are occurring (*e.g.* which components have failed) in a distributed system. Root cause analysis [129, 59] seeks to reconstruct those unknown events from limited monitoring data. Here we know exactly which events occurred, and instead seek to identify a minimal sequence of events.

**Bug Reproduction.** Record and replay techniques seek to reproduce a given concurrency bug [133, 134, 5, 96, 131, 123]. Once a bug has been reproduced, users can step through the (deterministic) execution and interactively examine the state of the system. Manually examining long system executions can be tedious, and our goal is to reduce such executions so that developers find it easier to identify the problematic code through replay or other means.

**Probabilistic Diagnosis.** Record and replay systems incur performance overheads at runtime that can be too prohibitive for production deployments. With the aim of avoiding the runtime overhead of deterministic replay, probabilistic diagnosis techniques [131, 94, 60] capture carefully selected diagnostic information (e.g. stack traces, thread & message interleavings) that should have high probability of helping developers find the root cause of a problem. One key insight underlying these techniques is cooperative debugging: the realization that even if one does not collect enough diagnostic information from a single bug report, it is quite likely that the bug will happen more than once so that diagnostic information can be correlated [73]. We assume more complete runtime instrumentation (during testing, not in production), but provide exact reproducing scenarios.

**Detecting Bugs in Production.** Despite our best efforts, bugs invariably make it into production. Still, we would prefer to discover these issues through means that are more proactive than user complaints. An old idea is useful here: distributed snapshots [19], a subset of the events in the system's execution such if any event e is contained in the subset, all 'happens-before' predecessors of e are also contained in the subset. Distributed snapshots allow us to obtain a global view of the state of all machines in the system, without needing to stop the world. Once we have a distributed snapshot in hand, we can

check assertions about the state of the overall system (either offline [77] or online [78]). Within the networking community, research along these lines has focused on verifying routing tables [62, 61, 85, 64] or forwarding behavior [137, 138]. We focus on reducing executions leading up to bugs, assuming we already have access to some mechanism for detecting those bugs.

**Instrumentation.** Making sense of unstructured diagnostic information pieced together from a large collection of machines is challenging. The goal of distributed tracing frameworks is to produce structured event logs that can be easily analyzed to understand performance and correctness bugs. The core idea [21] is simple: have the first machine assign an ID to the incoming request, and attach that ID (plus a pointer to the parent task) to all messages that are generated in response to the incoming request. Then have each downstream task that is involved in processing those messages log timing information associated with the request ID to disk. Propagating the ID across all machines results in a tree of timing and causality information.

Causal trees can be used[2] to: characterize the production workload [10], handle resource accounting [114] and 'what-if' predictions for resource planning [113], track flows across administrative domains [39], visualize traces and express expectations about how flows should be structured [100], monitor performance isolation in a multi-tenant environment [83], track the flow of packets through a network [50], and detect performance anomalies [102]. These use-cases enable developers to understand how, when, and where the system broke. In contrast, we seek to provide the minimal set of events needed for the software to violate an invariant.

[2]See [101] for a comprehensive survey of distributed tracing tradeoffs and use-cases.

# Chapter 6

# Concluding Remarks & Future Work

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 and concurrent systems are lagging behind their sequential counterparts. Inspired by the obvious utility of test case reduction tools for sequential executions, we sought to develop execution reduction techniques for distributed executions.

We investigated two scenarios where execution reduction could be applied to distributed systems: partially instrumented code, where executions are non-deterministic, and completely instrumented code, where we tie ourselves to a particular messaging library in order to precisely control all relevant sources of non-determinism. We applied our techniques to 7 different distributed systems, and our results leave us optimistic that these techniques can be successfully applied to a wide range of distributed and concurrent systems.

**Future Work.** We see several directions for future work on execution reduction. Some of these directions address limitations of the work presented in this dissertation, and others pose new questions.

- **Constrained Computational Models for Program Analysis.** Our execution reduction strategies are motivated by *program properties*, or constraints on the computational structure of the practical systems we care about in practice. We have yet to properly formalize those program properties. Can we formalize those program properties, and prove that practical systems exhibit those properties? Within those constrained computational models, can we prove that our execution reduction strategies achieve soundness and completeness?

  More generally, program analysis for arbitrary Turing machines and arbitrary pro-

gram properties is typically either undecidable or intractable. The distributed systems we build in practice however are not adversarial! Can we codify the properties of practical distributed systems that make them more easily amenable to program analysis? What program analysis tools can we provide to help alleviate the specific challenges (e.g. performance variability, concurrency) faced by distributed systems?

We list some of our conjectured program properties here:

*Reducibility:* Suppose that an invariant is defined over a small subset of each processes' local variables. Suppose further that, upon receiving a single message, each process will only update some of its local variables, not all of them. Consider reducing the overall state machine (defined as the cross product of the state machines of each process) to a machine that only contains state transitions whose updated local variables are within the domain of the invariant. It is likely that the path through this machine defined by the events in the original execution will contain *loops*. We can remove any events in the original execution that cause the machine to traverse those loops, and still arrive at the final unsafe state.

*Quiescence:* If the program p is quiescent – meaning that it is guaranteed to eventually stop sending messages – then the set of schedules to be explored is finite.

*Semi-Quiescence:* If the program p is semi-quiescent – meaning that it is quiescent except for some message type (e.g. heartbeat messages) – then the set of schedules to be explored is finite if you assume that the non-quiescent component of the program p is correct.

*K-Quiescence:* If the program p is k-quiescent – meaning that it is guaranteed to stop sending messages within k-steps, then the set of schedules to be explored is bounded.

*Commutativity:* If some of the events within the schedule space can be reordered without changing the outcome – i.e. p exhibits some degree of commutativity – then not all schedules need to be explored, viz. those where the commutative events are reordered. This is the intuition behind DPOR, but there are other forms of commutativity beyond the happens-before relation considered by DPOR.

*Symmetry:* Suppose that if we change the labels of some of the processes in the system, the behavior of the overall system is the same. We could then consider any executions that only differ in the placement of the labels as equivalent.

*Data Independence:* If the internal state transitions taken by p in response to external messages do not depend on the values (fields) of the external messages, then schedules with only differing values of the external messages can be considered equivalent.

*Bounded Edit Distance:* Consider the state machine defined as the cross product of all processes' state machines. Each edge is labeled with an event. Now consider all states that exhibit the invariant violation, and the k-shortest paths that lead up to any of those states. Bounded edit distance states that at least one of those paths has at most $\alpha$ transitions that were not in the original execution, and the remaining transitions can be converted to a subsequence of the original execution with at most $\beta$ reorderings, where $\alpha$ and $\beta$ are small constants.

*Recency of State:* A program p exhibits state recency if whenever the program p reaches quiescence, its memory footprint is bounded; that is, it forgets about events from before k steps of reaching quiescence.

*Monotonicity:* Let $f_p$ denote a function that takes a finite sequence of external events $E'$, and returns whether there exists a schedule $S'$ containing $E'$ that triggers the violation. A set of external events $E$ is monotonic if whenever $f_p(E') = true$, then for all subsequences $E''$ of $E'$, $f_p(E'') = true$.

- **Improving Upon Our Schedule Exploration Strategies.** In this dissertation we designed a small number of heuristics and empirically showed that they are effective. However, there may be other complementary and effective heuristics. Many of these new heuristics could in fact fall directly out of our formalization of program properties. A few examples:

  Exploring reorderings of events seems to be less fruitful than exploring unexpected events. Would it be effective to give DPOR a fixed budget of unexpected events to explore, and a smaller budget of reordering to explore?

  We have noticed that timers (messages sent to the node itself) seem to play a particularly important role in the control flow of the execution. Would it be effective to extend fixed budgets to 4 distinct budgets?: one for unexpected timers, one for reorderings of timers, one for unexpected messages, and one for reordered messages?

  Looking at visualizations of our executions, it becomes clear that there are "clusters" of events. Within those clusters, there are many happens-before edges, but across those clusters, there are few happens-before edges. Would either of these heuristics produce rapid reduction?: (i) prioritize reorderings within clusters before reorderings across clusters, or (ii) prioritize reorderings across clusters before reorderings within clusters?

  The 'provenance' of a message—the causal chain of message deliveries that preceded it—provides potentially useful information to reduction strategies. Can we cluster and prioritize messages according to their vicinity in the causal graph? Can we reason a priori about which internal events will *not* show up as a result of removing prior events?

- **Leveraging Whitebox Visibility.** In this dissertation we refrain from analyzing the code. But our refrain is by no means fundamental; can we apply static and dynamic analysis to improve our schedule exploration strategies? Can we gain lightweight visibility into internal state of the processes (e.g., through the invariant checking function we are already given by the application developer), without needing to tie ourselves to the details of a particular language or implementation?

- **Automatically Learning Schedule Exploration Strategies.** The schedule exploration strategies we developed here are designed by hand, based on our personal experiences debugging and understanding the programs under test. Could we instead have our tool *learn* appropriate schedule exploration strategies, as it experiments with more and more system behaviors? For example, could we infer a model of the program's state machine (a la Synoptic [13, 12]), and use this model to dynamically guide which schedules to explore next? We have already begun investigating this line of inquiry.

- **Synthesizing Message Fingerprints.** We currently define our message fingerprints by manually examining the message formats for all protocols used by the distributed system. This task may require a substantial amount of effort for some systems. Is is possible to automatically extract message fingerprints using program analysis? Or, can we experimentally distinguish between message field values that are non-deterministically generated, and message field values that are relevant to triggering invariant violations? (cf. Twitter's 'Diffy' Testing System [116]).

- **Providing Hints to Delta Debugging.** Delta debugging takes a domain-agnostic approach to selecting which input events to test (that is, it does not assume any knowledge of the structure or semantics of the inputs it seeks to reduce). However, the executions of distributed systems have rich structures (e.g., they are partially ordered by the happens-before relation). Can we improve delta debugging's complexity by providing it hints about this causal structure? For example, would we benefit from coercing delta debugging to split events along minimal cuts through the happens-before graph? Or, suppose we split events according to their locality or type rather than their time? Can we statically codify the semantics of input events to prevent delta debugging from exploring unfruitful event sequences?

- **Reducing Production Executions.** As we discussed in chapters 3.7 and 4.4, reducing production executions would require us to overcome several challenges. Can we obtain sufficient monitoring information from production without incurring prohibitive runtime overheads? Can we identify redundant events recorded in monitoring logs? Can we scale our test environment to match the size of production environments?

- **Choosing Appropriate Interposition Points.** We targeted OpenFlow (in chapter 3) and the Akka actor framework (in chapter 4) for a simple reason: thanks to the narrowness of these APIs, 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. By choosing appropriate interposition points, would it be possible to increase the generality and effectiveness of STS and DEMi, without substantially increasing engineering effort? Can execution reduction work equally well if applied to a lower layer of the stack (e.g. the network or syscall layer) rather than the application layer?

- **Detecting Bugs.** Our ultimate goal should be to prevent bugs from being deployed in the first place. In a sense, the techniques that we developed are simply strategies for *finding* problematic schedules within the intractably large schedule space. Can we apply these same strategies to improve concurrency testing, before debugging takes place? Since our scheduling strategies are based on small model property assumptions, can we build up test cases from small initial executions rather than generating them randomly? Can we use our knowledge of prior bugs to avoid finding new but uninteresting bugs that developers will not care to fix?

- **Preventing Configuration Errors.** Human configuration errors remain the largest source of production outages. More generally, developers and operators spend a significant portion of their time and effort dealing with configuration. To what extent can we design our systems to be autonomous and configuration-free? For the remainder of knobs we must expose to humans, what can we learn from the HCI community on how to design configuration interfaces to avoid errors and facilitate clear reasoning?

- **Designing Language Constructs for Safe Asynchronous Programming.** Our case studies have taught us much about the kinds of errors that developers of distributed systems make. These errors often involve asynchrony and partial failure, which are fundamental to networked systems. Yet, as our case studies demonstrate, asynchronous programming remains notoriously error prone. To what extent can we automatically synthesize tricky pieces of asynchronous code? Can we design language constructs, based on our understanding of these common mistakes, that side-step the need for developers to reason about the entire combinatorial space of possible message orderings?

Going forward, we see a pressing need to bring the state-of-the-art for distributed systems development tools up to speed with the more well studied and widely used programming languages and software engineering tools for sequential programs. We hope that continued interactions with researchers and practitioners alike will help us bridge this gap.

# Bibliography


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

[2] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. "Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication". In: International Workshop on Distributed Algorithms '97.

[3] *Akka official website*. http://akka.io/.

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

[5] Gautam Altekar and Ion Stoica. "ODR: Output-Deterministic Replay for Multicore Debugging". In: SOSP '09.

[6] Jason Ansel, Kapil Arya, and Gene Cooperman. "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop". In: IPDPS '09.

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

[8] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. "Testing Telecoms Software with Quviq QuickCheck". In: Erlang '06.

[9] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing Symbolic Execution with Veritesting". In: ICSE '14.

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

[11] Basho Blog. *QuickChecking Poolboy for Fun and Profit*. http://basho.com/posts/technical/quickchecking-poolboy-for-fun-and-profit/.

[12] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. "Inferring Models of Concurrent Systems from Logs of their Behavior with CSight". In: ICSE '14.

[13] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. "Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models". In: ESEC/FSE '11.

[14] Alan W Biermann and Jerome A Feldman. “On the Synthesis of Finite-State Machines from Samples of their Behavior”. In: IEEE ToC ’72.

[15] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. “A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs”. In: ASPLOS ’10.

[16] Martin Burger and Andreas Zeller. “Minimizing Reproduction of Software Failures”. In: ISSTA ’11.

[17] Yan Cai and WK Chan. “Lock Trace Reduction for Multithreaded Programs”. In: TPDS ’13.

[18] T.D. Chandra and S. Toueg. “Unreliable Failure Detectors for Reliable Distributed Systems”. In: JACM ’96.

[19] K. Mani Chandy and Leslie Lamport. “Distributed Snapshots: Determining Global States of Distributed Systems”. In: ACM TOCS ’85.

[20] Kai-hui Chang, Valeria Bertacco, and Igor L Markov. “Simulation-Based Bug Trace Minimization with BMC-Based Refinement”. In: IEEE TCAD ’07.

[21] Yen-Yang Michael Chen, Anthony Accardi, Emre Kiciman, David A Patterson, Armando Fox, and Eric A Brewer. “Path-Based Failure and Evolution Management”. In: SOSP ’04.

[22] J.D. Choi and A. Zeller. “Isolating Failure-Inducing Thread Schedules”. In: SIGSOFT ’02.

[23] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. “Flow-Sensitive Fault Localization”. In: VMCAI ’13.

[24] Koen Claessen and John Hughes. “QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs”. In: ICFP ’00.

[25] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. “Finding Race Conditions in Erlang with QuickCheck and PULSE”. In: ICFP ’09.

[26] James Clause and Alessandro Orso. “A Technique for Enabling and Supporting Debugging of Field Failures”. In: ICSE ’07.

[27] Holger Cleve and Andreas Zeller. “Locating Causes of Program Failures”. In: ICSE ’05.

[28] Katherine E Coons, Sebastian Burckhardt, and Madanlal Musuvathi. “GAMBIT: Effective Unit Testing for Concurrency Libraries”. In: PPoPP ’10.

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

[30] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. “ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay”. In: OSDI ’02.

[31] Mohamed A El-Zawawy and Mohammad N Alanazi. “An Efficient Binary Technique for Frace Simplifications of Concurrent Programs”. In: ICAST ’14.

[32] Alexander Elyasov, IS Wishnu B Prasetya, and Jurriaan Hage. “Guided Algebraic Specification Mining for Failure Simplification”. In: TSS ’13.

[33] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. “Dynamically Discovering Likely Program Invariants to Support Program Evolution”. In: IEEE ToSE ’01.

[34] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: JACM ’85.

[35] Cormac Flanagan and Patrice Godefroid. “Dynamic Partial-Order Reduction for Model Checking Software”. In: POPL ’05.

[36] *Floodlight Controller*. http://www.projectfloodlight.org/floodlight/.

[37] Floodlight FIXME comment. *Controller.java, line 605*. https://github.com/floodlight/floodlight/blob/2e9427e20ede7dc3941f8c15d2348bfcafdce237/src/main/java/net/floodlightcontroller/core/internal/Controller.java.

[38] Pedro Fonseca, Rodrigo Rodrigues, and Björn B Brandenburg. “SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration”. In: OSDI ’14.

[39] R. Fonseca, G. Porter, R.H. Katz, S. Shenker, and I. Stoica. “X-Trace: A Pervasive Network Tracing Framework”. In: NSDI ’07.

[40] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. “Frenetic: A Network Programming Language”. In: ICFP ’11.

[41] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. “Replay Debugging For Distributed Applications”. In: ATC ’06.

[42] *GNU’s guide to testcase reduction*. https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction.

[43] P Godefroid and N Nagappan. “Concurrency at Microsoft - An Exploratory Survey”. In: CAV ’08.

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

[45] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. "VL2: A Scalable and Flexible Data Center Network (Sec. 3.4)". In: SIGCOMM '09.

[46] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. "NOX: Towards an Operating System For Networks". In: CCR '08.

[47] Diwaker Gupta, Kenneth Yocum, Marvin Mcnett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. "To Infinity and Beyond: Time-Warped Network Emulation". In: NSDI '06.

[48] Mouna Hammoudi, Brian Burg, Gigon, and Gregg Rothermel. "On the Use of Delta Debugging to Reduce Recordings and Facilitate Debugging of Web Applications". In: ESEC/FSE '15.

[49] Nikhil Handigol, Brandon Heller, Vimal Jeyakumar, Bob Lantz, and Nick McKeown. "Reproducible Network Experiments Using Container Based Emulation". In: CoNEXT '12.

[50] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks". In: NSDI '14.

[51] Brandon Heller, Colin Scott, Nick McKeown, Scott Shenker, Andreas Wundsam, Hongyi Zeng, Sam Whitlock, Vimalkumar Jeyakumar, Nikhil Handigol, James McCauley, et al. "Leveraging SDN Layering to Systematically Troubleshoot Networks". In: HotSDN '13.

[52] Jeff Huang and Charles Zhang. "An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs". In: SAS '11.

[53] Jeff Huang and Charles Zhang. "LEAN: Simplifying Concurrency Bug Reproduction via Replay-Supported Execution Reduction". In: OOPSLA '12.

[54] John M Hughes. Personal Communication.

[55] John M Hughes and Hans Bolinder. "Testing a Database for Race Conditions with QuickCheck". In: Erlang '11.

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

[57] Nicholas Jalbert and Koushik Sen. "A Trace Simplification Technique for Effective Debugging of Concurrent Programs". In: FSE '10.

[58] Wei Jin and Alessandro Orso. “F3: Fault Localization for Field Failures”. In: ISSTA ’13.

[59] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. “Detailed Diagnosis in Enterprise Networks”. In: SIGCOMM ’09.

[60] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. “Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures”. In: SOSP ’15.

[61] Peyman Kazemian, Michael Change, Hongyi Zheng, George Varghese, Nick McKeown, and Scott Whyte. “Real Time Network Policy Checking Using Header Space Analysis”. In: NSDI ’13.

[62] Peyman Kazemian, George Varghese, and Nick McKeown. “Header Space Analysis: Static Checking For Networks”. In: NSDI ’12.

[63] Sepideh Khoshnood, Markus Kusano, and Chao Wang. “ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrency Bugs”. In: ISSTA ’15.

[64] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. “VeriFlow: Verifying Network-Wide Invariants in Real Time”. In: NSDI ’13.

[65] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. “An Overview of AspectJ”. In: ECOOP ’01.

[66] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. “Onix: A Distributed Control Platform for Large-scale Production Networks”. In: OSDI ’10.

[67] L. Zhang and L. Zhang and S. Khurshid. “Injecting Mechanical Faults to Localize Developer Faults for Evolving Software”. In: OOPSLA ’13.

[68] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: CACM ’78.

[69] Thomas D. LaToza, Gina Venolia, and Robert DeLine. “Maintaining Mental Models: a Study of Developer Work Habits”. In: ICSE ’06.

[70] Steven Lauterburg, Rajesh K Karmani, Darko Marinov, and Gul Agha. “Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques”. In: FASE ’10.

[71] Kyu Hyung Lee, Yunhui Zheng, Nick Sumner, and Xiangyu Zhang. “Toward Generating Reducible Replay Logs”. In: PLDI ’11.

[72] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. "SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems". In: OSDI '14.

[73] Ben Liblit. *Cooperative Bug Isolation*. UC Berkeley PhD Thesis, '07.

[74] Chia-Chi Lin, Virajith Jalaparti, Matthew Caesar, and Jacobus Van der Merwe. "DEFINED: Deterministic Execution for Interactive Control-Plane Debugging". In: ATC '13.

[75] Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. "MODIST: Transparent Model Checking of Unmodified Distributed Systems". In: NSDI '09.

[76] *Linux Kernel Containers*. linuxcontainers.org.

[77] Xuezheng Liu. "WiDS Checker: Combating Bugs in Distributed Systems". In: NSDI '07.

[78] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. "D$^3$S: Debugging Deployed Distributed Systems". In: NSDI '08.

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

[80] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. "Automatic Generation of Software Behavioral Models". In: ICSE '08.

[81] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. "Mining Invariants from Console Logs for System Problem Detection". In: ATC '10.

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

[83] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. "Retro: Targeted Resource Management in Multi-tenant Distributed Systems". In: NSDI '15.

[84] Nuno Machado, Brandon Lucia, and Luıs Rodrigues. "Concurrency Debugging with Differential Schedule Projections". In: PLDI '15.

[85] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. "Debugging the Data Plane with Anteater". In: SIGCOMM '11.

[86] James Mccauley. *POX: A Python-based OpenFlow Controller*. http://www.noxrepo.org/pox/about-pox/.

[87] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: Enabling Innovation in Campus Networks". In: SIGCOMM CCR '08.

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

[89] Madanlal Musuvathi and Shaz Qadeer. "Iterative Context Bounding for Systematic Testing of Multithreaded Programs". In: PLDI '07.

[90] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. "Finding and Reproducing Heisenbugs in Concurrent Programs". In: SOSP '08.

[91] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: ATC '14.

[92] ON.Lab. *Open Networking Operating System*. http://onlab.us/tools.html.

[93] *OpenStack Neutron*. https://wiki.openstack.org/wiki/Neutron.

[94] Sang Min Park. *Effective Fault Localization Techniques for Concurrent Software*. PhD Thesis, '14.

[95] Soyeon Park, Shan Lu, and Yuanyuan Zhou. "CTrigger: Exposing Atomicity Violation Bugs from their Hiding Places". In: ASPLOS '09.

[96] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H Lee, and Shan Lu. "PRES: Probabilistic Replay with Execution Sketching on Multiprocessors". In: SOSP '09.

[97] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. "Extending Networking into the Virtualization Layer". In: HotNets '09.

[98] *PULSE Tutorial*. http://www.erlang-factory.com/upload/presentations/191/EUC2009-PULSE.pdf.

[99] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. "Test-case Reduction for C Compiler Bugs". In: PLDI '12.

[100] Patrick Reynolds, Charles Killian, Janet L. Winer, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vadhat. "Pip: Detecting the Unexpected in Distributed Systems". In: NSDI '06.

[101] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. "So, you want to trace your distributed system? Key design insights from years of practical experience". In: CMU Tech Report, '14.

[102] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. "Diagnosing Performance Changes by Comparing Request Flows". In: NSDI '11.

[103] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. "Minimizing Faulty Executions of Distributed Systems". In: NSDI '16.

[104] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis, and Scott Shenker. "Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences". In: SIGCOMM '14.

[105] Ohad Shacham, Eran Yahav, Guy Golan Gueta, Alex Aiken, Nathan Bronson, Mooly Sagiv, and Martin Vechev. "Verifying Atomicity via Data Independence". In: ISSTA'14.

[106] Jiri Simsa, Randy Bryant, and Garth A Gibson. "dBug: Systematic Evaluation of Distributed Systems". In: SSV '10.

[107] Vijayaraghavan Soundararajan and Kinshuk Govil. "Challenges in Building Scalable Virtualized Datacenter Management". In: OSR '10.

[108] William Sumner and Xiangyu Zhang. "Comparative Causality: Explaining the Differences Between Executions". In: ICSE '13.

[109] Sriraman Tallam, Chen Tian, Rajiv Gupta, and Xiangyu Zhang. "Enabling Tracing of Long-Running Multithreaded Programs via Dynamic Execution Reduction". In: ISSTA '07.

[110] G. Tel. *Introduction to Distributed Algorithms*. Thm. 2.21. Cambridge University Press, 2000.

[111] Valerio Terragni, Shing-Chi Cheung, and Charles Zhang. "RECONTEST: Effective Regression Testing of Concurrent Programs". In: ICSE '15.

[112] *The ns-3 network simulator*. http://www.nsnam.org/.

[113] Eno Thereska and Gregory R Ganger. "Ironmodel: Robust Performance Models in the Wild". In: SIGMETRICS '08.

[114] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. "Stardust: Tracking Activity in a Distributed Storage System". In: SIGMETRICS '06.

[115] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. "Triage: Diagnosing Production Run Failures at the User's Site". In: SOSP '07.

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

[117] Rachel Tzoref, Shmuel Ur, and Elad Yom-Tov. "Instrumenting Where it Hurts: An Automatic Concurrent Debugging Technique". In: ISSTA '07.

[118] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. "Scalability and Accuracy in a Large-Scale Network Emulator". In: OSDI '02.

[119] Jie Wang, Wensheng Dou, Chushu Gao, and Jun Wei. "Fast Reproducing Web Application Errors". In: ISSRE '15.

[120] M. Weiser. "Program Slicing". In: ICSE '81.

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

[122] P. Wolper. "Expressing Interesting Properties of Programs in Propositional Temporal Logic". In: POPL '86.

[123] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. "OFRewind: Enabling Record and Replay Troubleshooting for Networks". In: ATC '11.

[124] Jifeng Xuan and Martin Monperrus. "Test Case Purification for Improving Fault Localization". In: FSE '14.

[125] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. "CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems". In: NSDI '09.

[126] Maysam Yabandeh and Dejan Kostic. *DPOR-DS: Dynamic Partial Order Reduction in Distributed Systems*. Tech. rep. EPFL, Tech Report 139173, 2009.

[127] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers". In: PLDI '11.

[128] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M Kirby. "Efficient Stateful Dynamic Partial Order Reduction". In: MCS '08.

[129] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. "A Survey of Fault Localization Techniques in Computer Networks". In: Science of Computer Programming '04.

[130] Xiaodong Yi, Ji Wang, and Xuejun Yang. "Stateful Dynamic Partial-Order Reduction". In: FMSE '06.

[131] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. “SherLog: Error Diagnosis by Connecting Clues from Run-time Logs”. In: ASPLOS ’10.

[132] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: NSDI ’12.

[133] C. Zamfir, G. Altekar, G. Candea, and I. Stoica. “Debug Determinism: The Sweet Spot for Replay-Based Debugging”. In: HotOS ’11.

[134] Cristian Zamfir and George Candea. “Execution Synthesis: A Technique For Automated Software Debugging”. In: EuroSys ’10.

[135] Andreas Zeller. “Yesterday, my program worked. Today, it does not. Why?” In: ESEC/FSE ’99.

[136] Andreas Zeller and Ralf Hildebrandt. “Simplifying and Isolating Failure-Inducing Input”. In: IEEE TSE ’02.

[137] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. “Automatic Test Packet Generation”. In: CoNEXT ’12.

[138] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. “Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks”. In: NSDI ’14.

[139] Sai Zhang and Congle Zhang. “Software Bug Localization with Markov Logic”. In: ICSE ’14.