Effective Randomized Concurrency Testing with Partial Order Methods

Xinhao Yuan

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2020

ABSTRACT

Effective Randomized Concurrency Testing with Partial Order Methods

Xinhao Yuan

Modern software systems have been pervasively concurrent to utilize parallel hardware and perform asynchronous tasks. The correctness of concurrent programming, however, has been challenging for real-world and large systems. As the concurrent events of a system can interleave arbitrarily, unexpected interleavings may lead the system to undefined states, resulting in denials of services, performance degradation, inconsistent data, security issues, etc. To detect such concurrency errors, concurrency testing repeatedly explores the interleavings of a system to find the ones that induce errors. Traditional systematic testing, however, suffers from the intractable number of interleavings due to the complexity in real-world systems. Moreover, each iteration in systematic testing adjusts the explored interleaving with a minimal change that swaps the ordering of two events. Such exploration may waste time in large homogeneous sub-spaces leading to the same testing result. Thus on real-world systems, systematic testing often performs poorly to reveal even simple errors within a limited time budget. On the other hand, randomized testing samples interleavings of the system to quickly surface simple errors with substantial chances, but it may as well explore equivalent interleavings that do not affect the testing results. Such redundancies weaken the probabilistic guarantees and performance of randomized testing to find any errors.

Towards effective concurrency testing, this thesis leverages partial order semantics with randomized testing to find errors with strong probabilistic guarantees. First, we

propose partial order sampling (POS), a new randomized testing framework to sample interleavings of a concurrent program with a novel partial order method. It effectively and simultaneously explores the orderings of all events of the program, and has high probabilities to manifest any errors of unexpected interleavings. We formally proved that our approach has exponentially better probabilistic guarantees to sample any partial orders of the program than state-of-the-art approaches. Our evaluation over 32 known concurrency errors in public benchmarks shows that our framework performed 2.6 times better than state-of-the-art approaches to find the errors. Secondly, we describe Morpheus, a new practical concurrency testing tool to apply POS to high-level distributed systems in Erlang. Morpheus leverages dynamic analysis to identify and predict critical events to reorder during testing, and significantly improves the exploration effectiveness of POS. We performed a case study to apply Morpheus on four popular distributed systems in Erlang, including Mnesia, the database system in standard Erlang distribution, and RabbitMQ, the message broker service. Morpheus found 11 previously unknown errors leading to unexpected crashes, deadlocks, and inconsistent states, demonstrating the effectiveness and practicalness of our approaches.

Contents

List of Figures **iv**

List of Tables **vii**

Acknowledgements **x**

Introduction **1**

Chapter 1 Preliminary **8**

1.1 Concurrent Systems 9

1.2 Event Labeling 10

1.3 Concurrency Errors 11

1.4 Partial Orders of Schedules 13

Chapter 2 Background **15**

2.1 Systematic Concurrency Testing 15

2.2 Randomized Concurrency Testing 17

Chapter 3 Partial Order Sampling **23**

3.1 Probabilistic Error-Detection Guarantees 23

3.2 Running Example 24
3.3 The POS Framework 25
3.4 The Partial Order Method of POS 30
3.5 Implementation 41
3.6 Evaluation 42
3.7 Summary 47

Chapter 4 Effective Concurrency Testing For Distributed Systems 50

4.1 Erlang Background 50
4.2 Overview 53
4.3 Implementation 63
4.4 Errors Found 66
4.5 Evaluation 76
4.6 Related Work 83
4.7 Summary 85

Chapter 5 Making Lock-free Data Structures Verifiable with Artificial Transactions 86

5.1 Overview 88
5.2 Implementation 91
5.3 Evaluation 95
5.4 Summary 102

Conclusion 104

Bibliography **106**

List of Figures

Figure 1.1 An example program running two concurrent threads $\{p, q\}$ with local variable `a` for thread p and shared variable `x` (initially is 0). 10

Figure 1.2 An example program with two concurrent threads with the shared variables `x` and `y` (initially set to 0). Statement 4 will fail if it is executed after statement 2 but before statement 3. 12

Figure 2.1 An example program illustrating random walk's weak guarantee of error detection. Variable `x` is initially 0, and any statement `step(...)` does not access `x`. 18

Figure 2.2 An example illustrating PCT's redundancy in exploring the partial orders of the program. Assume each statement is atomic. The statements after `barrier()` are enabled only after both threads executed the `barrier()` statements. 20

Figure 2.3 An example program illustrating the exponential bias of RAPOS. Thread p (q) has k steps accessing `x` (`y`) before accessing `z`. Variables `x`, `y`, and `z` are initially 0. 22

Figure 3.1 The running example. 24

Figure 3.2 An adversary program for random walk to have the worst error-detection guarantee 40

Figure 3.3 Conflict rates in macro benchmarks in ascending order over 1,000 trials in POS* scheduling. Rates are measured as $\frac{\max\{\#\text{ of conflicts}\}}{\max\{\#\text{ of events}\}\times(\max\{\#\text{ of threads}\}-1)}$. `fsbench_bad` is ignored since all its run crashes immediately. 46

Figure 4.1 The workflow of Morpheus. 55

Figure 4.2 An inconsistent view error in `gen_leader`, a leader election protocol implemented in Erlang. "$\rightarrow$ ..." denotes the result of its preceding operation. 56

Figure 4.3 POS can sample the order of `A`$\rightarrow$`B` with high probability that depends on the priorities assigned to `A`, `B`, and $\texttt{E}_i$, despite that there are numerous other x, y, and z operations. 56

Figure 4.4 `receive` pattern and its transformed matching function. The branch of the `receive` is taken only if the process receives a tuple with two elements, where the first element is symbol `reply`, and the second the element (assigned to `R`) is not symbol `error`. 64

Figure 4.5 The test scenario used to find `locks-1`. Three client processes concurrently acquire three locks in different orderings. For example, client 1 will lock in the ordering of `[1]`, `[2]`, and `[3]`. 68

Figure 4.6 The deadlock caused by prematurely version resetting. `x` and `y` are names of locks. 70

Figure 4.7 The atomicity violation in server probing. "$\rightarrow$ ..." denotes the result of its preceding operation. 71

Figure 4.8 The sync code for `mnesia` schema change, where the message of fault notification "{`mnesia_down, Node`}" is also treated as an acknowledgment of the commit. 74

Figure 4.9 Runtime performance of Morpheus on `gproc-1` and `mnesia-2` over 100 iterations. The numbers on the right show the decomposition of runtime (except common). 83

Figure 5.1 A lock-free stack (a) and a failure-causing test case (b). 89

Figure 5.2 The architecture of Txit. Placement plans are synthesized into a proposed program, which is profiled and checked, and which feeds into new placement plans. 92

Figure 5.3 Number of schedules (in log scale) vs transaction size. The horizontal line is at 10^4, and any result below this line is exact. 97

Figure 5.4 Microbenchmark for evaluating TSX overhead. 100

List of Tables

Table 3.1 Partial-order coverage on the micro benchmark programs. Columns under "benchmark" are program characteristics explained in Section 3.6.1. "$0(x)$" represents incomplete coverage. "Geo-mean" shows the geometric means of coverage, but conservatively accounts any incomplete coverage as $\frac{1}{5\times10^7}$ 44

Table 3.2 Partial-order coverage on the micro benchmark programs with 50% read events. POS*- means POS* ignoring read-read independence. 45

Table 3.3 Bug hit ratios on macro benchmark programs. "Geo-mean" shows the geometric means of hit ratios, but conservatively accounts any no-hit case as $\frac{1}{t}$, where t is the number of trials on the case. 49

Table 4.1 Core concurrency primitives in Erlang. 52

Table 4.2 Summary of the distributed systems tested and errors founded by Morpheus 67

Table 4.3 Error-detection comparison on chain replication protocol. The "Systematic" column shows the number of iterations for systematic testing to find the error for each case. The "Random walk" and "POS" columns show the average number of iterations for the algorithms to find the error in each case. 78

Table 4.4 Error-detection performance of different randomized testing algorithm. "+" means with conflict analysis. "**Mean**" row summarizes each of algorithms with the geometric means. "**Ratio to POS+**" row shows the ratio of overall performance compared to POS+. "**CA Improv.**" row shows the improvements of conflict analysis on the average performance for all algorithms. The summary of random walk is not available due to the missed errors. 79

Table 4.5 Results of conflict analysis with different signature schemes on `gproc-1` and `mnesia-2`. "Hit-ratio" denotes the error-finding performance of Morpheus without and with conflict analysis. "FNs" denotes falsely ignored conflicting operations, and "FPs" denotes falsely explored non-conflicting operations. 82

Table 5.1 Evaluated lock-free data structures. 96

Table 5.2 Normalized execution time of the test cases with transactions over without (smaller is better). The baseline column shows the normalized execution time at the starting point of the search for each data structure when every operation is made a transaction. 98

Table 5.3 Normalized execution time of single-threaded test cases with transactions over without (smaller is better). 99

Table 5.4 Comparison of load and store instruction cycles under different conditions. 101

Acknowledgements

The work presented in this dissertation would not have been possible without help and support of many people.

My greatest gratitude to my advisor, Professor Junfeng Yang, who selflessly guided me throughout my Ph.D. This dissertation would not have been completed without your invaluable support, encouragement, and inspirations.

Special thanks to Professors Jason Nieh and Ronghui Gu, and Doctors Madan Musuvathi and Franjo Ivancic. They served in my thesis committee and provided insightful and constructive suggestions to improved this dissertation.

I would also like to thank Professor Simha Sethumadhavan, and my colleagues Yang Tang, Gang Hu, David Williams-King, and Lingmei Weng. They worked with me and gave me valuable suggestions during my Ph.D. on research projects, academic writing, and many other aspects.

Last but not least, I wanted to thank my wife Hong Zhou, who supported me physically, financially, and mentally over the thousands of the days and nights of my reckless adventure.

To my beloved parents and my wife.

Introduction

Modern software systems have been pervasively concurrent to utilize parallel hardware and perform asynchronous tasks. Reasoning about the correctness of concurrent systems, however, is notoriously difficult. The behavior of a concurrent system is determined by the arbitrary interleaving of events among its components. It is extremely hard to consider all the possible interleavings when building the system, and an unexpected interleaving may manifest a so-called *concurrency error*, leading the system into undefined states, and causing potentially serious consequences, such as crashes, performance degradation, inconsistent states, security issues.

To detect concurrency errors of a system, *concurrency testing* takes a scenario of the system, controls the interleaving of its events , and explores all the possible interleavings to find the ones that induce concurrency errors. Various strategies can be used for exploration, and two classes of strategies have been extensively studied: *systematic concurrency testing* [28, 48, 71, 30, 37] (SCT) leverages model checking to exhaustively enumerate interleavings from the interleaving space. One fundamental challenge for model checking (and thus SCT) is the rapidly growing size of the interleaving space, exponential to the execution length of the test. Moreover, the innate depth-first search order of SCT makes tiny changes to interleavings explored, e.g., flip the ordering of two events, and favors

the changes towards the end of interleavings. Such an exploration may get stuck in a homogeneous subspace of interleavings that produce the same or similar outcomes without revealing errors.

Towards mitigating the challenges, *randomized concurrency testing* [58, 14, 44] (RCT) randomly samples the interleaving space, simultaneously exploring every part of the interleavings with simple strategies. RCT is lightweight and easy to deploy in real-world scenarios. Empirical studies showed that RCT effectively found errors in real-world concurrent systems. This finding matches the small scope hypothesis that real-world errors are non-adversarial, and often can be manifest if a few events happen in the right order, which RCT has good probabilities to achieve.

Despite the empirical effectiveness, unexpected biases may apply to RCT strategies, preventing them to surface even trivial errors. Random walk, a basic RCT strategy that randomly schedules any enabled events, may degrade exponentially to sample an ordering of two events. Probabilistic concurrency testing (PCT) algorithm improved upon random walk, to sample orderings of a few events with significant probabilistic guarantees. Having said that, PCT does not consider the *partial order semantics* of events – it may redundantly sample different orderings of *independent* events, whose ordering does not matter to the execution result. Such redundancies in PCT waste its testing resource and weaken its probabilistic guarantees.

Thesis Topic. Motivated by the limitations of existing RCT approaches, we aim to *leverage partial order semantics to improve RCT for better theoretical guarantees and empirical performance of error-finding* – the major challenge is the inherent gap between the exist-

ing partial order methods and the characteristics of RCT. *Partial order reduction* [29, 25, 2, 4, 9] (POR) techniques have been widely used in SCT to avoid redundant exploration for orderings of independent events. POR techniques dictate the depth-search order of the exploration, conflicting with the randomized exploration. Moreover, POR techniques often rely on heavy bookkeeping, negating the lightweight that makes RCT applicable to test complex systems.

Contribution 1: Partial Order Sampling Framework. Towards bridging the fundamental gap, we broke new ground to apply partial order semantics to RCT with a novel framework, *partial order sampling* (POS). The basic framework is extremely simple and general: it assigns events with independently random priorities, and always executes the event with the highest priorities among all available events. Such general rule of scheduling allows precise and formal analysis of sampling orderings of events based on the comparison of the random priorities of events. Based on the analysis, we identified a bias towards sampling different orderings of dependent events. To reduce the bias, we invented a new partial order method, POS*, by reassigning priority of an event when scheduling another dependent event – we proved that the method uniformly samples any partial orders of a concurrent system with exponentially better probabilities than state-of-the-art RCT approaches. To our knowledge, POS* is the first RCT approach that leverages partial order semantics with probabilistic guarantees. As empirically measured in our micro benchmarks, POS* brought 134.1$\times$ better guarantees to sample any partial orders compared to random walk and PCT. On error-finding performance, we measured POS* in real-world multi-threading programs, including Firefox's Javascript engine. POS*

performed 2.6× better than other approaches to detect the concurrency errors.

Contribution 2: Effective Concurrency Testing for Distributed Systems. Towards effective concurrency testing for real-world concurrent systems, we built Morpheus, the first RCT tool that leverages POS to test distributed systems, cornerstones for modern civilization in the era of cloud computing. The major challenge of testing distributed systems is their sheer complexity increase from multi-threaded programs. Distributed systems need to handle not only local concurrency as in multi-threaded programs, but also asynchronous communication across nodes to function correctly in unreliable distributed environments. Moreover, since nodes in a distributed systems have only partial views of the system, their communication is inevitably verbose to maintain the global safety and liveness of the system. Such inter-node communication further expands to layers of implementation details of low-level operating system primitives.

Morpheus tackles this challenge with two ideas. First, Morpheus uses the history of explored executions to predict what events may affect the partial order semantics of the test in future executions. If an event always occur independently with other events in any interleavings, Morpheus can ignore it in further explorations without lowering the guarantee of finding any errors, and this idea effectively accelerates error detection. Morpheus represents the history of explored executions in a succinct form whose size is proportional to the size of the program, incurring only minor bookkeeping. Results show that it is highly accurate based on only a few historic executions. This idea benefits not only POS, but RCT approaches in general.

Secondly, to avoid the implementation complexity in distributed systems, Morpheus

targets high-level program languages designed for distributed programming. Modern programming languages and frameworks increasingly provide first-class concurrency support that simplifies the development of concurrent systems, such as Go [62], Erlang [23], Mace [34], and P# [21]. For instance, Erlang has been well-known for its simple yet expressive support of distributed programming, such as messaging and fault-tolerance in and across distributed nodes. It is widely adopted in large-scale distributed services such as Amazon SimpleDB [6] and the backend of WhatsApp [67]. By targeting higher-level language and frameworks, Morpheus avoid being buried within the massive number of events generated by low-level layers in a distributed system, and greatly boosts the chance of finding intriguing protocol-level concurrency errors.

We implemented Morpheus for distributed systems written in Erlang. It leverages program rewriting to intercept Erlang communication primitives and explore their orderings, requiring no modifications to a checked system. Morpheus itself is written in Erlang, with the exception of 50 lines of modifications to the Erlang runtime. Morpheus properly isolates itself from a checked system so that messages in Morpheus and the checked system cannot interfere. This isolation also enables Morpheus to run multiple virtual nodes on the same physical node, further simplifying checking. Morpheus provides a virtual clock to check timer behaviors and speed up testing. Whenever an error is found, Morpheus stores a trace for deterministic replay of the error.

We evaluated Morpheus on four popular distributed systems in Erlang including RabbitMQ, a message broker service, and Mnesia, a first-party distributed database in the standard distribution of Erlang. Results showed that Morpheus is effective. It found new errors in every system, 11 total, all of which are flaws in the core protocol, and will cause

deadlocks, unexpected failures, or inconsistencies of the system. The conflict analysis effectively improved the error-detection performance of Morpheus by up to 415.71%.

Contribution 3: Making Lock-free Data Structures Verifiable with Artificial Transactions. Besides testing large-scale concurrent systems, we also studied the verification problem for a particular class of concurrent programs — lock-free data structures. Comparing to traditional concurrent programming with synchronization objects such as locks, lock-free data structures are considered more lightweight and scalable because they do not cause context switches. Reasoning about of lock-free data structures, however, is significantly harder because of their fine-grained and ad-hoc synchronizations, and they have much larger space of possible interleaving than traditional concurrent programs. On the other hand, lock-free data structures are often used in performance critical components such as operating system kernels (e.g. using RCU [69] mechanism), making their correctness crucial to the reliability of the entire system.

To address the challenge of verifying lock-free data structures, we presented Txit, a system that greatly reduce the set of interleavings by introducing artificial transactions in their implementations. Txit leverages hardware support of transactional memory to enforce these transactions in runtime. Evaluation on six popular lock-free data structure libraries showed that Txit made it easy to verify them while incurring acceptable runtime overhead. Further analysis showed that two inefficiencies in the Intel Haswell hardware support contributed the most to this overhead.

Thesis Structure. The rest of the thesis is organized as follows. Chapter 1 introduces preliminary definitions and formal models. Chapter 2 studies related work on concurrency testing, including original analysis results for previous work. Chapter 3 presents the partial order sampling framework with both formal analysis and evaluation. Chapter 4 describes Morpheus, including its technical details and implementation, and studies the concurrency errors found. Chapter 5 describes Txit. Finally we conclude.

Chapter 1

Preliminary

In this chapter, we first formalize concurrent systems and schedules (Section 1.1). We then introduce event labeling to represent schedules as events sequences (Section 1.2). We define concurrency errors and characterize them with event orderings (Section 1.3). Finally, we introduce partial orders of schedules based on the event independence (Section 1.4).

Basic Notations

We use ":=" as "defined as" and "=" as equality. We denote a sequence s with n ordered elements as $\langle e_1, e_2, \ldots, e_n \rangle$ (with ϵ representing the empty sequence $\langle\rangle$). Let $|s| := n$ and $s[i] := e_i$. Let $s +\!\!+ s'$ denote the concatenation of s before s', and $s \bullet e := s +\!\!+ \langle e \rangle$. Denote $e \in s$ if $\exists\, i\ ,\ \ s[i] = e$. Sequence s' is a *prefix* of s if there exists a sequence s^* s.t. $s' +\!\!+ s^* = s$. Let $s[i \ldots j]$ denote the sub-sequence of $\langle s[i], s[i+1], \ldots, s[j] \rangle$. For any set Σ, let Σ^* denote the set of all finite sequences whose elements are in Σ.

We use upper-cased symbols for functions and mappings, and lower-cased symbols for others. Specially, we use calligraphic letters for a few global constants.

1.1 Concurrent Systems

Concurrency in a software system denotes the fact that different components, i.e. processes, of the system executing transitions without specifying their ordering. From a high level, a process performs either deterministic and local transitions based on the current state of the process, or transitions on shared objects to exchange data and synchronize its execution with others. The behavior of the system is thus determined by the ordering of such communications.

We formally define a concurrent system as a state machine. Given a concurrent system with finite processes identified by labels in $\mathcal{P}$, a state of the state machine consists of the local states of the processes and the shared objects of the concurrent system. Any behavior of the concurrent system corresponds to a path of transitions of the state machine, starts with the unique initial state, transiting by scheduling a process $p \in \mathcal{P}$ for each step. When a process p is scheduled, it executes the current communication of p, followed by the local computation in the process until the next communication of p is about to execute. Note that not all processes can be scheduled in a given state, as some communication is partially defined on the states of the shared objects. For example, when a process is about to acquire a lock, the process can only be scheduled when the lock is released. We define $\mathcal{S} \subset \mathcal{P}^*$ as the set of finite sequences of processes that are possible to schedule from the initial state. Each sequence $s \in \mathcal{S}$, called a *schedule*, represents a possible behavior of the system, leading the state machine to a unique state by scheduling the processes in order. Define $\mathrm{En}(s) := \{p \mid s \bullet p \in \mathcal{S}\}$ as the *enabled* processes at s that can be scheduled. A schedule s is *complete* if $\mathrm{En}(s) = \varnothing$, otherwise it is *partial*.

Thread p	Thread q
`a = x;`	`x = 1;`
`x = a + 1;`	

Figure 1.1: An example program running two concurrent threads $\{p, q\}$ with local variable `a` for thread p and shared variable `x` (initially is 0).

Example 1.1. Consider the multi-threaded program presented in Figure 1.1. The program starts with the two threads[a] with identifiers $\{p, q\}$ communicating on the shared variable `x`, which is initially 0. One complete schedule of the program is $\langle p, q, p \rangle$, which first schedules thread p, then thread q, and finally thread p. After the three steps, both threads terminate, and the states of variables are `a = 0; x = 1`.

[a] In convention, we use the term "threads" for concurrent components in the context of multi-threaded programs.

Assumptions. We assume that (1) all memory accesses we discuss follow the *sequential consistency* model, as operations in relaxed models can be translated into sequential consistent operations without observable difference, and (2) the space of schedules $\mathcal{S}$ is finite. These assumptions are reasonable and common in concurrency testing work.

1.2 Event Labeling

To represent how the steps transit the state in the schedule, we map each step of the schedule to *events* with *event labeling*:

Definition 1.2. An *event labeling* E is a tuple $(\mathcal{L}_E, \mathrm{P}_E, \mathrm{N}_E)$ where $\mathcal{L}_E$ is a set of *events*, $\mathrm{P}_E : \mathcal{L} \to \mathcal{P}$, and $\mathrm{N}_E : \mathcal{S} \times \mathcal{P} \to \mathcal{L}_E$. $\mathrm{N}_E(s, p)$ maps the next step of process p after s to an event, such that the following conditions are met:

Affinity. $\forall\, s\ p\ ,\ \ s \in \mathcal{S} \Rightarrow \mathrm{P}_E(\mathrm{N}_E(s, p)) = p$

Uniqueness. $\forall\, s\ s'\ p\ , s \in \mathcal{S} \wedge s +\!\!+ s' \in \mathcal{S} \wedge \mathrm{N}_E(s, p) = \mathrm{N}_E(s +\!\!+ s', p) \Rightarrow p \notin s'$

Stability. $\forall\, s\ p\ p'\ ,\ \ p \neq p' \wedge s \bullet p' \in \mathcal{S} \Rightarrow \mathrm{N}_E(s, p) = \mathrm{N}_E(s \bullet p', p)$

To refer to an event inside a schedule, for $s \in \mathcal{S}$ and $i \in [1, |s|]$, let $\mathrm{L}_E(s, i) := \mathrm{N}_E(s[1 \ldots i-1], s[i])$ be the event of the i-th step in the schedule. Specially, let $\mathrm{N}_E(s) := \{\mathrm{N}_E(s, p)\}$ and $\mathrm{L}_E(s) := \{\mathrm{L}_E(s, i)\}$. From now on, we ignore the subscription $\cdot_E$ when E is clear in the context.

The abstract definition of event labeling allows flexible realizations to encode the semantics of steps. For example, for multi-threaded programs, an event labeling can encode the shared variables accessed by the event. Nevertheless, one of the simplest event labeling is

- $\mathcal{L} := \{(p, k) \mid p \in \mathcal{P}, k \in \mathbb{N}\}$
- $\mathrm{N}(s, p) := (p, k)$ where $k = |\{i \mid s[i] = p\}|$

which maps any step of scheduling p to (p, k) after p has been scheduled for k times. Any additional semantics can be encoded by extending this labeling.

1.3 Concurrency Errors

We formalize a concurrency error $\mathcal{E} \subset \mathcal{S}$ as the set of complete interleavings reaching any observerable erroneous states, such as assertion failures and deadlocks. A concurrency error can be hard to represent, as it is comprised of potentially long schedules. With event labeling, we can represent the error by sets of ordering constraints of events (event

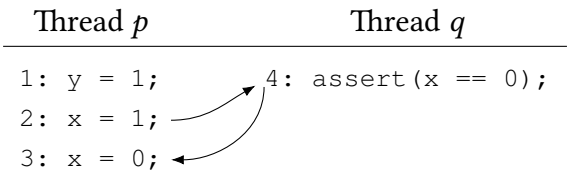

Figure 1.2: An example program with two concurrent threads with the shared variables x and y (initially set to 0). Statement 4 will fail if it is executed after statement 2 but before statement 3.

ordering for short) implying the error. Define that $s \in \mathcal{S}$ *follows* an *event ordering* $O \subseteq \mathcal{L} \times \mathcal{L}$ if

$$\forall\, a\ b\ i\ j\ ,\ (a,b) \in O \wedge \mathrm{L}(s,i) = a \wedge \mathrm{L}(s,j) = b \Rightarrow i < j$$

An event ordering O *implies* an error $\mathcal{E}$ if every complete schedule following O is in $\mathcal{E}$, and there always exists a complete schedule following O from a partial schedule following O.

Example 1.3. Consider the multi-threaded program in Figure 1.2. Statement 4 asserts the value read from x is 0, otherwise the program fails. All complete schedules for this program are

$$\{\langle p,p,p,q\rangle\ ,\langle p,p,q\rangle\ ,\langle p,q,p,p\rangle\ ,\langle q,p,p,p\rangle\}$$

The failure of statement 4 is a concurrency error of $\{\langle p,p,q\rangle\}$. If we use the statement labels as event labeling, the complete schedules can be mapped into sequences of events:

$$\{\langle 1,2,3,4\rangle\ ,\langle 1,2,4\rangle\ ,\langle 1,4,2,3\rangle\ ,\langle 4,1,2,3\rangle\}$$

The event ordering $\{(2,4),(4,3)\}$ implies the error.

Previous work [47, 14, 38] studied concurrency errors in context of multi-threaded programs and distributed systems, suggesting that the errors can be implied with the orderings of few events. The finding matches the small scope hypothesis [33, Section 5.1.3] that real-world errors tends to have simple root causes.

The structures of event orderings are used to categorize the implied concurrent errors. For examples, errors implied by two ordered events $\{(a, b)\}$ are called *order violations*; errors implied by three ordered events $\{(a, b), (b, c)\}$ are called *atomicity violations*, if a and c belong to the same process.

1.4 Partial Orders of Schedules

Given a schedule s of the program, the final state of s is determined by the ordering of its events. If two adjacent events are *independent*, swapping their ordering in the schedule will not change the final state. Formally, the independence of events is a symmetric relation $I \in \mathcal{L} \times \mathcal{L}$, such that if $(a, b) \in I$, $\mathrm{P}(a) \neq \mathrm{P}(b)$, and any schedule of the form:

$$s_1 := s +\!\!+ \langle p, q \rangle +\!\!+ s'$$

with $\mathrm{N}(s, p) = a$ and $\mathrm{N}(s \bullet p, q) = b$ must lead to the same state and have the same events by swapping their ordering, resulting in a *partial-order equivalent* schedule:

$$s_2 := s +\!\!+ \langle q, p \rangle +\!\!+ s'$$

Denote $s_1 \simeq s_2$ if s_1 can be turned into s_2 by swapping adjacent and independent events, and therefore s_1 and s_2 must both reach an error state or not.

In Example 1.3, event `1` is independent with event `4`, as they are accessing different global variables. Generally, events are independent if they always operate on different resources, such as memory accesses of different locations, messages deliveries to different recipients, etc. A fine-grained independence may include memory reads on the same location, as such reads must read the same result regardless of the ordering to each other. From now on, we assume an event independence I is given if the context is clear.

Denote a pair of events *conflicting* if they are not independent and may occur in both orderings in schedules. For any schedule, only the orderings of conflicting events may affect its final state. We denote the *partial order* of schedule s regarding I as the minimum event ordering $\mathrm{PO}(s) \subseteq 2^{\mathcal{L}\times\mathcal{L}}$, such that:

1. $\forall\, a\ b\ i\ j\ ,\ (a,b) \notin I \wedge \mathrm{L}(s,i) = a \wedge \mathrm{L}(s,j) = b \wedge i < j \Rightarrow (a,b) \in \mathrm{PO}(s)$
2. $\forall\, a\ b\ ,\ (a,b) \in \mathrm{PO}(s) \Rightarrow (b,a) \notin \mathrm{PO}(s)$
3. $\forall\, a\ b\ c\ ,\ (a,b) \in \mathrm{PO}(s) \wedge (b,c) \in \mathrm{PO}(s) \Rightarrow (a,c) \in \mathrm{PO}(s)$

The partial order of a schedule represents the essential ordering of events for reaching its final state. For any two schedules s and s', $s \simeq s'$ if and only if $(L(s) = L(s') \wedge \mathrm{PO}(s) = \mathrm{PO}(s'))$.

Chapter 2

Background

In this chapter, we study previous work closely related to concurrency testing. We first revisit systematic concurrency testing and its reduction and bounding techniques (Section 2.1). Then we review state-of-the-art randomized concurrency testing approaches (Section 2.2) with their strengths and weaknesses.

2.1 Systematic Concurrency Testing

Systematic concurrency testing (SCT) exhaustively enumerates all complete schedules of $\mathcal{S}$ based on the depth-first search (DFS). SCT repeatedly explores the system for multiple iterations. Each iteration follows the three phases to explore one complete schedules:

Backtracking. When an iteration begins, SCT selects the *longest* schedule s_{pre} from the set of unexplored schedules $\mathcal{S}_{bt}$ (initially $\{\epsilon\}$), and continue to replaying.

Replaying. At the current schedule s, SCT chooses to schedule a process p s.t. $s \bullet p$ is a prefix of s_{pre}. SCT keeps replaying until no such p exists, and then continues to exploring.

Exploring. SCT schedules any $p \in \text{En}(s)$, and add any sibling schedule $s \bullet p'$ s.t. $p' \in \text{En}(s) \setminus \{p\}$ to $\mathcal{S}_{bt}$ for further backtracking. SCT keeps exploring until the current

schedule is complete.

Each iteration in SCT explores a new schedule if there are any schedules left unexplored in $\mathcal{S}_{bt}$, otherwise all schedules have been explored, and the test is finished. Given enough testing time, SCT always find a concurrency error, or finish the test to verify the system (as aforementioned, $\mathcal{S}$ is assumed finite).

The major challenges of SCT are twofold. First, its often impossible for SCT to finish the test for real-world systems due to the rapidly growing size of $\mathcal{S}$, potentially exponential to the length of any complete schedule. Conceptually, consider an ideal concurrent system with n processes, where each process is comprised of k steps to schedule. The total size of $\mathcal{S}$ is $|\mathcal{S}| = (nk)!/(k!)^n = O(n^{nk}/k^n)$, which quickly exceeds the number of schedules affordable to test for real-world systems. Secondly, because of the DFS order, SCT must explore all schedules with a common prefix before exploring a schedule with a different prefix. It is thus difficult for SCT to find even a simple error if its prefix is not explored in the beginning of the test.

2.1.1 Reduction and Bounding Techniques

Towards making SCT tractable for large programs, reduction techniques reduce the number of schedules to explore by skipping redundant schedules that lead to already explored states. State-of-the-art reduction techniques include partial order reduction [29, 25, 2, 4, 9, 51, 3] that avoids partial-order equivalent schedules, symmetry reduction [19, 61] that exploits state symmetry, and interface reduction [30] that caches states by interface behaviors. These techniques can often provide exponential reduction on the schedule

space for real-world programs, but the reduced space still grows fast for complex systems. Moreover, reduction techniques usually dictate the depth-first search order, leaving the exploration bias unsolved, and involve heavy overhead for bookkeeping the explored schedules.

Bounding techniques restrict the SCT exploration to the subspace of the schedules within complexity bounds, motivated by the finding of simple root causes of real-world errors. For example, preemption bounding [47] bounds the number of *preemptions* in schedules. Formally, the i-th step of schedule s is a preemption if

$$s[i] \neq s[i-1] \wedge s[i-1] \in \mathrm{En}(s[1 \ldots i-1])$$

Preemption bounding prevents SCT from exploring any schedule with the number of preemptions more than a given bound. Other bounding techniques include delay bounding [22] that bounds the number of "deviations" from a deterministic scheduler, and backtrack-bounding [71, 8] that bounds the number of backtracking performed to reach a schedule. With a bounded schedule space, however, SCT still suffers from the bias of the search order.

2.2 Randomized Concurrency Testing

Randomized concurrency testing (RCT) samples schedules with randomized strategies. RCT strategies are often simple and lightweight, using only random variables instead of heavy bookkeeping in SCT. RCT tackles the search order problem of SCT by simulta-

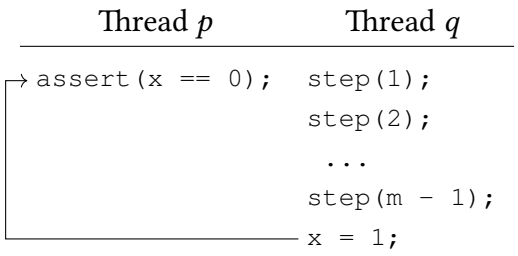

Figure 2.1: An example program illustrating random walk's weak guarantee of error detection. Variable `x` is initially 0, and any statement `step(...)` does not access `x`.

neously and randomly exploring every part of the schedules, but giving up verification. Instead of verification, RCT provides probabilistic guarantees to sample any schedules, and often has good probabilities to sample schedules that manifest simple root causes of a few ordered events. Studies [63] showed that RCT outperformed SCT in finding the errors in real-world multi-threaded programs.

We now revisit state-of-the-art randomized strategies, understand their strengths and limitations. We show that existing RCT approaches do not consider partial order semantics with probabilistic guarantees, and thus can be highly biased to sample some partial orders and ignore some others

2.2.1 "Naïve" Random Walk

Random walk independently chooses a random enabled process to schedule at every step. Despite its simplicity, the major limitation of random walk is the exponential bias for sampling different orderings of two events.

Example 2.1. Consider Figure 2.1. The assertion in thread `A` fails if and only if it runs after "`x = 1`" in thread q. Without a prior knowledge of which order between

the two statements fails the assertion, both orders should be sampled uniformly because the probabilistic guarantee of detecting the error is the *minimum* probability to manifest both of the two orderings. Unfortunately, random walk may yield extremely non-uniform sampling probabilities. In this example, to trigger the failure, random walk has to skip the assertion m times, yielding a probabilistic guarantee of only $1/2^m$ to trigger the assertion failure.

2.2.2 Probabilistic Concurrency Testing

Inspired by the simple root causes of the real-world errors, probabilistic concurrency testing [14] (PCT) algorithm uses a priority-based scheduling strategy to manifest simple event orderings with probabilistic guarantees. PCT takes three explicit parameters, n, the maximum number of processes in the system, k, the maximum length of schedules, and d the number of events to sample for different orderings. The algorithm works as followings:

1. In the beginning of an iteration, PCT initializes distinct random priorities from $[1, n]$ for all processes, and selects $d-1$ random steps numbers $k_1, \ldots, k_{d-1}$ from $\mathcal{U}\{1, k\}$. These step numbers are used to delay next events by changing the corresponding process priorities.
2. After initializing priorities and step numbers, PCT schedules the enabled process with the highest priority for each step.
3. Every time after a process is scheduled, PCT checks if the current step number (initially 1) matches any k_i. If any k_i is matched, PCT reassigns the priorities of the scheduled process to $-i$ and removes k_i from the selected numbers. After the check,

Thread p	Thread q
`1: x++;`	`5: y--;`
`2: barrier();`	`6: barrier();`
`3: y++;`	`7: x = 0;`
`4: x--;`	`8: y = 1;`

Figure 2.2: An example illustrating PCT's redundancy in exploring the partial orders of the program. Assume each statement is atomic. The statements after `barrier()` are enabled only after both threads executed the `barrier()` statements.

PCT increases the current step number by 1.

As proven in [14], by manipulating process priorities with the randomly initialized priorities, PCT is able to make a random set of d events happen after any other events, and arrange the d events in any ordering (if possible in the system) with the probability lower-bounded by

$$\frac{1}{nk^{d-1}}$$

The probabilistic guarantees are strong for errors implied by orderings of a few events, which requires a small d for PCT to surface. The probabilistic guarantees of PCT, however, depend on complete parameters of n, k and d, which could be difficult for programs (and errors) without domain knowledge. Previous work suggested estimating n and k by profiling, weakening the formal completeness of the guarantees.

Despite the strong probabilistic guarantees, PCT does not consider the partial order of events entailed by the concurrent system, such that the explored schedules are still quite redundant. Consider the example in Figure 2.2. Assume each statement is atomic, and the statements after `barrier()` are enabled only after both `barrier()` are executed. Both statement `1` and `5` are executed before the barrier and do not conflict with

any statement. Statements 3 and 8 form a conflict, and so do statements 4 and 7. Depending on how each conflict is ordered, the program schedules have total three different partial orders. Without considering the effects of barriers, however, PCT (with $d = 2$) will attempt to change priorities after 1 or 5 in vain, which makes no difference in the partial order. Furthermore, without considering independent events, PCT may first test an event sequence of

$$\langle 1, 2, 5, 6, 3, 7, 8, 4 \rangle$$

by changing priorities after 3 with initial priorities $p \mapsto 2, q \mapsto 1$, and then test a partial-order equivalent and thus redundant event sequence

$$\langle 5, 6, 1, 2, 7, 3, 4, 8 \rangle$$

by changing priorities after 7 with initial priorities $p \mapsto 1, q \mapsto 2$. Such redundancies in PCT waste testing resources and weaken the probabilistic guarantee.

2.2.3 Random Partial Order Sampling (RAPOS)

Towards uniformly sampling the partial orders of concurrent systems, Sen [58] proposed RAPOS algorithm. Although it shares the same goal as in this thesis, RAPOS has no probabilistic guarantees of sampling partial orders. For each iteration, RAPOS works as follows. RAPOS schedules a concurrent system in *batches* of steps, with each batch executing a set of independent events. At the beginning of a batch, RAPOS maintains a *schedulable* set of events, which is a subset of the currently available events. RAPOS randomly selects a

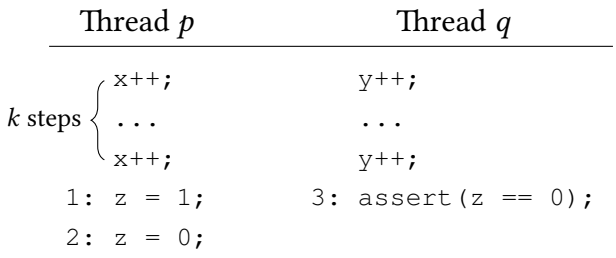

Figure 2.3: An example program illustrating the exponential bias of RAPOS. Thread p (q) has k steps accessing `x` (`y`) before accessing `z`. Variables `x`, `y`, and `z` are initially 0.

scheduled set of events from the schedulable events, such that all events in the scheduled set are independent to each other. With the scheduled set randomly decided, RAPOS performs a batch to schedule all the scheduled events in any order. After the batch, the new schedulable set is the currently available events that are dependent to any of the scheduled events in the last batch. If no such events exist, RAPOS randomly selects the set of an available event to be the new schedulable set. RAPOS repeatedly executes batches until the schedule is complete.

It is proved [58] that RAPOS is able to sample any partial order of a program, by constructing the sets of events to schedule. It is possible, however, to construct examples, such that RAPOS is exponentially biased to sample certain partial orders. In the example shown in Figure 2.3, there are only three partial orders depending on how `z` is accessed by p and q. To sample the assertion failure such that the event `3` is scheduled between `1` and `2`, RAPOS has to select the desired scheduled sets of events for k batches. For each batch the scheduled set must contain both `x++;` in p and `y++;` in q, which happens in probability of (at most) $1/2$. Thus RAPOS samples the error (and its partial order) in probability of at most $1/2^{k+2}$.

Chapter 3

Partial Order Sampling

In this chapter, we introduce our work of partial order sampling (POS), the first RCT approach that samples partial orders of programs with strong probabilistic guarantees. We first define the formal guarantees for a RCT approach to detect any error of a program in the worst case. Then we set up a running example to demonstrate the guarantees for different RCT approach to find its error. Next, we present the basic framework with formal analyses, introduce the novel priority reassignment to uniformly sample different partial orders, and demonstrate the exponentially improved probabilistic guarantees. Finally, we describe the implementation of POS, and show our evaluation results to demonstrate that POS achieved better empirical error-finding performance than state-of-the-art approaches.

3.1 Probabilistic Error-Detection Guarantees

As aforementioned, the partial orders of schedules partition the schedule space of a system into equivalent classes. Each class of partial order may lead to potentially incorrect final state different to other classes. Therefore, any possible partial order has to be explored. For a RCT strategy that randomly explores the complete schedules of a system with random variables, denote $\mathrm{Sample}(R)$ as the explored schedule with input R as the random

```
global int x = y = z = w = 0;
```

Thread p	Thread q
	`local int a = b = 0;`
`1: x++;`	`5: x = 1;`
`2: y++;`	`6: a = x;`
`3: signal(w);`	`7: y = a;`
`4: assert(z < 5);`	`8: wait(w);`
	`9: b = y;`
	`10: z = a + b;`

Figure 3.1: The running example.

variables. The *minimum* probability the strategy to sample any partial orders of complete schedules in a system are called its *probabilistic error-detection guarantee*, denoted as

$$min_{s \in \mathcal{S} \wedge \mathrm{En}(s)=\varnothing} \Pr[\mathrm{Sample}(R) \simeq s]$$

3.2 Running Example

Figure 3.1 shows the running example for comparing the probabilistic guarantees of different RCT approaches. In this example, all shared variables (e.g., `x`, `w`, etc.) are initialized to 0. The program consists of two threads, i.e., p and q. Thread q will be blocked at `8` by `wait(w)` until `w` > 0. Thread p will set `w` to be 1 at `3` via `signal(w)` and unblock thread q. The assertion at `4` will fail if, and only if, the program is scheduled according to the following sequence of events:

$$\langle 5, 1, 6, 7, 2, 3, 8, 9, 10, 4 \rangle$$

To detect this error, random walk has to make the correct choice at every step. Among

all ten steps, three of them only have a single option: 2 and 3 must be executed first to enable 8, and 4 is the only statement left at the last step. Thus, the probability of random walk to reach the error is $1/2^7 = 1/128$. As for PCT, we have to change the priorities after statements 5 and 1 among ten statements, and thread q must have higher initial priority than p. Thus the probability for PCT is $1/2 \times 1/10 \times 1/10 = 1/200$.

As we will explain later, POS with the partial order method can detect this error with a substantial probability of $1/48$, much higher than other approaches. Indeed, our formal guarantees ensure that any behavior of this program can be tested with probabilities of at least $1/60$.

3.3 The POS Framework

The basic framework of POS is extremely simple. Given a concurrent system and its event labeling, POS takes a random event priority mapping $\mathrm{Pri} : \mathcal{L} \to \mathbb{R}$, that maps any event to a priority independently following the uniform distribution $\mathcal{U}(0, 1)$. According to the priority mapping, POS always schedule the process with the event mapped to the highest priority among all enabled processes. The abstract algorithm is shown in Algorithm 1.

To understand the probabilistic guarantees of POS, We first present analyses based on the notion of *guides*. We then establish the formal connection of POS to other RCT approaches. Finally, we study the bias in POS that sample different ordering of conflict events, resulting in the sub-optimal error-detection guarantees.

Algorithm 1 Sample a schedule $s \in \mathrm{L}$ with POS under the random priority mapping Pri

procedure $\mathrm{Sample}_{\mathrm{POS}}(\mathrm{Pri})$
    $s \leftarrow \epsilon$
    while $\mathrm{En}(s) \neq \varnothing$ **do**
        $p^* \leftarrow \arg\max_{p \in \mathrm{En}(s)} \mathrm{Pri}(\mathrm{N}(s, p))$
        $s \leftarrow s \bullet p^*$
    end while
    return s
end procedure

3.3.1 Probabilistic Analysis

To understand how POS can manifest any concurrency errors with probabilistic guarantees, we first formulate the notion of *guides*, and consider how POS follows a guide to manifest errors.

A guide g is an acyclic event ordering $g \subseteq 2^{\mathcal{L} \times \mathcal{L}}$ that constraints the next steps of any schedules by the orderings among the available events. Formally, denote

$$\mathrm{En}(s, g) := \{p \mid p \in \mathrm{En}(s), \forall\, p' \ ,\ p' \in \mathrm{En}(s) \Rightarrow (\mathrm{N}(s, p'), \mathrm{N}(s, p)) \notin G\}$$

For a schedule $s \in \mathcal{S}$, denote $s \nsim g$ if any step of s schedules the limited enabled process in g, formally:

$$\forall\, i \ ,\ s[i] \in \mathrm{En}(s[1 \ldots i-1], g)$$

A guide g *leads to* error $\mathcal{E}$ if:

$$\forall\, s \ ,\ s \in \mathcal{S} \wedge s \nsim g \wedge \mathrm{En}(s, g) = \varnothing \Rightarrow s \in \mathcal{E}$$

Let $\mathrm{Scope}(g) = \{e \mid (e', e) \in g \vee (e, e') \in g\}$ and $\mathrm{Delayed}(g) = \{e \mid (e', e) \in g\}$ be

the number of events constrained by G and potentially blocked by G, respectively. The following theorem shows the probabilistic guarantees of POS.

Theorem 3.1. *For any concurrency error $\mathcal{E}$ and a guide G that leads to the error. Let $c = \mathrm{Scope}(g)$ and $d = \mathrm{Delayed}(g)$. The following holds*

$$\Pr[\mathrm{Sample}_{\mathrm{POS}}(\mathrm{Pri}) \in \mathcal{E}] \geq \frac{(c-d)!}{c!} \geq \frac{1}{c^d}$$

Proof. Since g leads to $\mathcal{E}$, we consider the probability of how POS returns a complete schedule s s.t. $s \nvdash g$. A sufficient condition for that is letting Pri map the priorities of events ordered as they are constrained in g, i.e.

$$\forall\ a\ b\ ,\ (a,b) \in g \Rightarrow \mathrm{Pri}(a) > \mathrm{Pri}(b)$$

Since g is acyclic, and the priority for any event is independently random. Pri can satisfy the sufficient condition by mapping priorities of $\mathrm{Delayed}(g)$ to be lower than priorities in $\mathrm{Scope}(g) \setminus \mathrm{Delayed}(g)$, and letting the ordering of priorities of $\mathrm{Delayed}(g)$ satisfy any linearization of $G \cap (\mathrm{Delayed}(g) \times \mathrm{Delayed}(g))$, which must exist. Thus the lower-bounded probability to satisfy the sufficient condition is

$$\frac{(c-d)!}{c!}$$

3.3.2 Sampling Partial Orders of Schedules

Theorem 3.1 shows the formal guarantees for sampling any set of schedules implied by a guide. It also applies to sampling any schedules with the partial order of a given schedule s^*. To simplify our model, we assume that any $\mathrm{N}(s^*, p)$ is never enabled in any prefix of s^*. Such assumption is valid if all processes eventually enter terminal states in complete schedules. Under this assumption, $\mathrm{PO}(s^*)$ can be a guide for POS to sample partial-order equivalent schedules of s thanks to the follow lemmas:

Lemma 3.2.

$$\forall\, s\ p\ i\ j,\ \ s \in \mathcal{S} \wedge p \in \mathrm{En}(s[1 \dots i], \mathrm{PO}(s)) \wedge j \in [i+1, |s|]$$
$$\Rightarrow (\mathrm{L}(s, j), \mathrm{N}(s[1 \dots i], p)) \notin \mathrm{PO}(s)$$

Proof. Let s, p, i, j be one of the smallest counterexample regarding to j. Let $e := \mathrm{N}(s[1 \dots i], p)$ and $e' := \mathrm{L}(s, j)$. There must exist $i + 1 < j' < j$, such that $(\mathrm{L}(s, j'), \mathrm{L}(s, j)) \in \mathrm{PO}(s)$. Otherwise $e' = \mathrm{N}(s[1 \dots i], \mathrm{P}(e'))$ must hold, contradicting with $p \in \mathrm{En}(s[1 \dots i], \mathrm{PO}(s))$. According to the construction of $\mathrm{PO}(s)$, $(\mathrm{EI}(s, j'), \mathrm{N}(s[1 \dots i], p)) \in \mathrm{PO}(s)$ must hold. It contradicts with the smallest j, as $j' < j$ is also a counterexample. Thus the original lemma holds.

Lemma 3.3.

$$\forall\, s\ ,\ s \not\sim \mathrm{PO}(s^*) \Rightarrow (\exists\, s^+\ ,\ s^+ \simeq s^* \wedge s^+[1 \dots |s|] = s)$$

Proof. Use induction on $|s|$. When $|s| = 0$ it is trivial. When $|s| > 0$, find $\bar{s}'$ and p such that $\bar{s}' \bullet p = \bar{s}$. According to inductive assumption, we can find an $\bar{s}^+$ such that

$$\bar{s}^+ \simeq s^* \wedge \bar{s}^+[1 \dots |\bar{s}'|] = \bar{s}'$$

Since $p \in \mathrm{En}(\bar{s}', \mathrm{PO}(\bar{s}^+))$ (notice that $\mathrm{PO}(\bar{s}^+) = \mathrm{PO}(\bar{s}^*)$), $\mathrm{N}(\bar{s}', p)$ must occurs as $\mathrm{L}(\bar{s}^+, i)$ for some $i > |\bar{s}|$. According to Lemma 3.2, all events $\mathrm{L}(\bar{s}^+, i')$ with $i' \in [|\bar{s}|, i-1]$ are independent with $\mathrm{L}(\bar{s}^+, i)$. Thus, we can construct $\bar{s}^*$ from $\bar{s}^+$ such that:

$$\bar{s}^* \simeq \bar{s}^+ \wedge \bar{s}^*[1 \dots |\bar{s}|] = \bar{s}$$

Corollary 3.4.

$$\forall\, s\,,\ (\mathrm{En}(s) = \varnothing \wedge s \mid\!\sim \mathrm{PO}(s^*)) \Leftrightarrow s \simeq s^*$$

Although $\mathrm{PO}(s^*)$ may *not* be the minimum guide leading to the partial order of s^*, the lower bound in Theorem 3.1 can be reached. Let $k := |s^*|$ and $n := |\mathcal{P}|$, if (almost) all events of s need to be delayed after some other conflicting events, the basic POS may only have the probability of $1/k!$ to sample such partial order. For random walk the probability can be $1/n^k$, which is exponentially better than POS when $n \ll k$.

The factorial degradation leads us to locate the major bias of POS: the *propagated comparison* of priorities. To follow the guide of a partial order, an event e delaying after a large set of events needs to have a very low priority, since the priority has to be lower than the priorities of events it delays for. Thus, for any conflicting event e' delayed after e, $\mathrm{Pri}(e')$ must be even lower than Pri, which is hard to satisfy for a random $\mathrm{Pri}(e')$.

Here, we explain how basic POS samples the error described in Figure 3.1 with a bias. To sample the error, the priority map has to satisfy the following constraints:

$$\mathrm{Pri}(5) > \mathrm{Pri}(1) \wedge \mathrm{Pri}(1) > \mathrm{Pri}(6) \wedge \mathrm{Pri}(6) > \mathrm{Pri}(2) \wedge \mathrm{Pri}(7) > \mathrm{Pri}(2)$$
$$\wedge \mathrm{Pri}(8) > \mathrm{Pri}(4) \wedge \mathrm{Pri}(9) > \mathrm{Pri}(4) \wedge \mathrm{Pri}(10) > \mathrm{Pri}(4)$$

Note that these are also the necessary constraints for POS to follow sample the error. The probability that a random Pri satisfies the constraints is $1/120$. The propagated comparison of priorities can be illustrated by the first three steps in the scheduling:

$$\mathrm{Pri}(5) > \mathrm{Pri}(1) > \mathrm{Pri}(6) > \mathrm{Pri}(2)$$

which happens in the probability of $1/24$. On the other hand, however, random walk can sample these three steps in the probability of $1/8$.

3.4 The Partial Order Method of POS

We now show how to reduce the bias of POS by eliminating the propagated comparison of priorities to follow a given partial order $\mathrm{PO}(s^*)$ of schedule s^*. During the sampling procedure of Algorithm 1, suppose the current schedule is s and POS is choosing the next process to schedule at line 4. Consider any process p s.t. $p \notin \mathrm{En}(s, \mathrm{PO}(s^*))$, that is, scheduling p will diverge the current partial order from s^*. Suppose there is a p' such that $p' \in \mathrm{En}(s, \mathrm{PO}(s^*)) \wedge p \in \mathrm{En}(s \bullet p', \mathrm{PO}(s^*))$. If we reset the priority of $\mathrm{N}(s, p)$ right after scheduling p', all the priority comparison causing the delay of $\mathrm{N}(s, p)$ will not be

propagated to any further event e such that $(\mathrm{N}(s, p), e) \in \mathrm{PO}(s)$.

However, there is no way for POS to exactly know when to reset a priority of event e during the sampling, since $\mathrm{PO}(s^*)$ is unknown and not provided. Notice that

$$\forall\, s\ p\ p'\ ,\ s \vdash\!\!\sim \mathrm{PO}(s^*) \wedge p \notin \mathrm{En}(s, po(s^*)) \wedge$$
$$p' \in \mathrm{En}(s, \mathrm{PO}(s^*)) \wedge p \in \mathrm{En}(s \bullet p', \mathrm{PO}(s^*)) \Rightarrow (\mathrm{N}(s, p), \mathrm{N}(s, p')) \notin I$$

If we reset the priority of all enabled events are not independent with $\mathrm{N}(s, p')$ after scheduling p', the propagated comparison of priorities can be eliminated, at the cost that the priority of an event may be reset for multiple times.

Probabilistic Analysis of Priority Reassignment

We now analyze in probabilistic guarantee of POS with the priority reassignment, named POS* for short, follows the partial order of a schedule $\mathrm{PO}(s^*)$. We first model the priority changes into the sampling procedure by introducing event indices. Then we derive the sufficient conditions of the sampling procedure to produce a partial-order equivalent schedule of s^*. Finally we analyze the probability of satisfying the sufficient conditions with random priority (re-)assignments.

Event indicies. To give a formal and precise analysis, we introduce the notion of *event indices* for any events in a schedule s:

$$\mathrm{EI}(s, i) := |\{j \mid j < i \wedge (\mathrm{L}(s, j), \mathrm{L}(s, i)) \notin I \wedge s[i] \neq s[j]\}|$$

Events in the same process are not counted into the indices for brevity. Also denote the indices of available events at s as

$$\mathrm{EIN}(s, p) := |\{i \mid (\mathrm{L}(s, i), \mathrm{N}(s, p)) \notin I\}|$$

By defining event indices, we model the priority changes into indexed priorities with the priority mapping expanded to $\mathrm{Pri} : \mathcal{L} \times \mathbb{N} \to \mathbb{R}$, as shown in Algorithm 2. Every time a process is scheduled, the priorities of available events will change if and only if they are conflicting to the scheduled event, since their indices increase on conflicts.

We here show how the priority reassignment improves POS in the example of Figure 3.1. We construct the sufficient condition of Pri for $\mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri})$ to sample the error as following:

$$\begin{aligned}\mathrm{Pri}(\mathtt{5}, 0) > \mathrm{Pri}(\mathtt{1}, 0) \wedge \mathrm{Pri}(\mathtt{1}, 1) > \mathrm{Pri}(\mathtt{6}, 0) \wedge \mathrm{Pri}(\mathtt{6}, 1) > \mathrm{Pri}(\mathtt{2}, 0)\\ \wedge \mathrm{Pri}(\mathtt{7}, 0) > \mathrm{Pri}(\mathtt{2}, 0) \wedge \mathrm{Pri}(\mathtt{8}, 1) > \mathrm{Pri}(\mathtt{4}, 0)\\ \wedge \mathrm{Pri}(\mathtt{9}, 0) > \mathrm{Pri}(\mathtt{4}, 0) \wedge \mathrm{Pri}(\mathtt{10}, 0) > \mathrm{Pri}(\mathtt{4}, 0)\end{aligned}$$

Compared to the constraints in the basic POS framework, there is no propagated comparison of indexed priorities. Since each $\mathrm{Pri}(e, i)$ is independently random following $\mathcal{U}(0, 1)$, the probability of Pri satisfying the constraints is $1/2 \times 1/2 \times 1/3 \times 1/4 = 1/48$.

Sufficient conditions of POS* following $\mathrm{PO}(s^*)$. We now analyze the conditions of the indexed priorities in which POS* follow $\mathrm{PO}(s^*)$. In fact, the indices of scheduled

Algorithm 2 Sample a schedule $s \in \mathrm{L}$ with POS with priority reassignment

procedure $\mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri})$
  $s \leftarrow \epsilon$
  while $\mathrm{En}(s) \neq \varnothing$ **do**
    $p^* \leftarrow \arg\max_{p \in \mathrm{En}(s)} \mathrm{Pri}(\mathrm{N}(s, p), \mathrm{EIN}(s, p))$
    $s \leftarrow s \bullet p^*$
  end while
  return s
end procedure

events are invariants in partial-order equivalent schedules, which represent their partial order. Denote $\mathrm{EI}(s) := \{(\mathrm{L}(s, i), \mathrm{EI}(s, i))\}$. We have

$$\forall\, s\ ,\ \mathrm{EI}(s) = \mathrm{EI}(s^*) \Leftrightarrow s \simeq s^*$$

A simple sufficient condition is having priorities $\mathrm{Pri}(e, i)$ s.t. $(e, i) \in \mathrm{EI}(s^*)$ higher than any other indexed priorities in the procedure, so that $\mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri})$ always schedule any process with the event and index matching $\mathrm{EI}(s^*)$. To find out the number of other indexed priorities in the sampling procedure, we define *direct conflicts*. Events e and e' is a *direct conflict* in schedule s if they are not independent, and e is scheduled in when e' is available, formally:

$$\exists\, i\ p,\ \mathrm{L}(s, i) = e \wedge s[i] \neq p \wedge \mathrm{N}(s[1 \ldots i-1], p) = e' \wedge (e, e') \notin I$$

Each direct conflict of e and e' corresponds to a new indexed priority of e', thus the number of other indexed priorities is the number of direct conflicts in the sampled schedule s. Denote

$$\mathrm{Confl}(s^*) := max_{s \simeq s^*} |\{(e, e') \mid (e, e') \text{ is a direct conflict in } s\}|$$

Since each scheduled event in s can have at most $|\mathcal{P}| - 1$ direct conflicts, we have

$$\mathrm{Confl}(s^*) \leq (|\mathcal{P}| - 1)\,|s^*|$$

Consider the numeric ordering of the priorities in POS*. Let $to_{\mathrm{Pri}} \subseteq 2^{\mathrm{EI}(s^*)\times\mathrm{EI}(s^*)}$ be a total order of $\mathrm{EI}(s^*)$. Denote that a priority mapping Pri satisfies to_{Pri}, or $\mathrm{Pri} \circ\!\!- to_{\mathrm{Pri}}$, if

$$\forall\, e\ i\ e'\ i'\ ,\ ((e,i),(e',i')) \in to_{\mathrm{Pri}} \Leftrightarrow \mathrm{Pri}(e,i) > \mathrm{Pri}(e',i')$$

The following lemma shows that $\mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri})$ is fixed for any Pri, given the Pri follows the total order of to_{Pri}, $\mathrm{PO}(s^*)$, and the result of sampling follows $\mathrm{PO}(s^*)$.

Lemma 3.5. Let Pri_1 and Pri_2 be any priority mappings, and let $s_1 := \mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri}_1)$ and $s_2 := \mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri}_2)$. If

$$\mathrm{Pri}_1 \circ\!\!- to_{\mathrm{Pri}} \wedge s_1 \simeq s^* \wedge \mathrm{Pri}_2 \circ\!\!- to_{\mathrm{Pri}} \wedge s_2 \simeq s^*$$

then $s_1 = s_2$.

Proof. We show that under the conditions of the lemma, both s_1 and s_2 are uniquely determined by to_{Pri}, despite any difference between Pri_1 and Pri_2. Consider s_1. since $s_1 \simeq s^*$, we know that for every step in $\mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri}_1)$, the intermediate schedule s in Line 4 of Algorithm 2 must follow s^*. The eligible p in Line 4 to extend s to continue following s^* must have the pending event indexed accordingly in $\mathrm{EI}(s^*)$. Since Pri_1 is ordered according to to_{Pri}, the indexed event with the highest priority among eligible

Algorithm 3 Constructing $\mathrm{PO}_{\mathrm{EI}}(s^*, to_{\mathrm{Pri}})$ based on $\mathrm{Sample}_{\mathrm{POS}*}$

procedure CONSTRUCT-$\mathrm{PO}_{\mathrm{EI}}(s^*, to_{\mathrm{Pri}})$
  $s \leftarrow \epsilon$
  $po \leftarrow \varnothing$
  while $\mathrm{En}(s) \neq \varnothing$ **do**
    assert$(s \nsim \mathrm{PO}(s^*))$
    $c_{\mathrm{Sch}} \leftarrow \{(\mathrm{N}(s, p), \mathrm{EIN}(s, p)) \mid p \in \mathrm{En}(s)\}$
    $c_{\mathrm{Follow}} \leftarrow c_{\mathrm{Sch}} \cap \mathrm{EI}(s^*)$
    assert$(c_{\mathrm{Follow}} \neq \varnothing)$
    $p^* \leftarrow p \mid (\mathrm{N}(s, p), \mathrm{EIN}(s, p))$ is the maximum of c_{Follow} ordered by to_{Pri}
    $po \leftarrow po \cup \{((\mathrm{N}(s, p^*), \mathrm{EIN}(s, p^*)), ei) \mid ei \in c_{\mathrm{Sch}} \setminus c_{\mathrm{Follow}}\}$
    $s \leftarrow s \bullet p^*$
  end while
  assert$(s \simeq s^*)$
  return po
end procedure

p is thus determined by to_{Pri}, independent to Pri_1. The same independence holds for Pri_2. Thus $s_1 = s_2$.

Review the proof above, it shows that given to_{Pri} fixed, POS* must schedules among a fixed set of indexed events to follow $\mathrm{PO}(s^*)$. Algorithm 3 constructs a unique partial order between such indexed events, expanded from to_{Pri}, namely $\mathrm{PO}_{\mathrm{EI}}(s^*, to_{\mathrm{Pri}})$, such that

$$\forall\, \mathrm{Pri}, \mathrm{Pri} \circ\!\!- \mathrm{PO}_{\mathrm{EI}}(s^*, to_{\mathrm{Pri}}) \Leftrightarrow (\mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri}) \simeq s^* \wedge \mathrm{Pri} \circ\!\!- to_{\mathrm{Pri}}) \tag{3.1}$$

Take $s^* = \langle 5, 1, 6, 7, 2, 3, 8, 9, 10, 4\rangle$ as an example to construct $\mathrm{PO}_{\mathrm{EI}}(s^*, to_{\mathrm{Pri}})$. Here

$$\mathrm{EI}(s^*) = \{(5, 0), (1, 1), (6, 1), (7, 0), (2, 1), (3, 0), (8, 1), (9, 2), (10, 0), (4, 1)\}$$

If we take to_{Pri} by the numerical ordering of the event labels, which, e.g., contains

$((1,1),(5,0))$ and $((2,1),(3,0))$,

$$\begin{aligned}\mathrm{PO}_{\mathrm{EI}}(s^*, to_{\mathrm{Pri}}) = to_{\mathrm{Pri}} &\cup \{((5,0),(1,0)),((1,1),(6,0)),((6,1),(2,0))\} \\ &\cup \{((7,0),(2,0)),((8,1),(4,0)),((9,1),(4,0))\} \\ &\cup \{((10,0),(4,0))\}\end{aligned}$$

Probabilistic lower bound of POS*. We are now ready to derive the probabilistic lower bound of POS* to follow a given schedule s^* in partial-order. We partition the space of priority mappings by their numeric ordering among $\mathrm{EI}(s^*)$. For each such total order to_{Pri}, Lemma 3.6 shows the probabilistic lower bound of a random Pri s.t. $\mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri}) \simeq s^*$.

Theorem 3.6. *Given* $to_{\mathrm{Pri}} \subseteq 2^{\mathrm{EI}(s^*)\times\mathrm{EI}(s^*)}$,

$$\Pr[\mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri}) \simeq s^* \mid \mathrm{Pri} \circ\!\!- to_{\mathrm{Pri}}] \geq \left(\frac{1}{|\mathcal{P}|}\right)^{|s^*|} r^u$$

where

$$r := |\mathcal{P}|\,|s^*|\,/(|s^*| + (|\mathcal{P}|-1)\lceil \mathrm{Confl}(s^*)/(|\mathcal{P}|-1)\rceil)$$

$$u := (|s^*| - \lceil \mathrm{Confl}(s^*)/(|\mathcal{P}|-1)\rceil)/2$$

Proof. According to (3.1), the probability in the lemma equals to

$$\frac{\Pr[\mathrm{Pri} \circ\!\!- \mathrm{PO}_{\mathrm{EI}}(s^*, to_{\mathrm{Pri}})]}{\Pr[\mathrm{Pri} \circ\!\!- to_{\mathrm{Pri}}]} = |s^*|! \Pr[\mathrm{Pri} \circ\!\!- \mathrm{PO}_{\mathrm{EI}}(s^*, to_{\mathrm{Pri}})]$$

For simplicity, let $o := \mathrm{PO}_{\mathrm{EI}}(s^*, to_{\mathrm{Pri}})$ in this proof. The proof includes three parts: We first analyze the structure characteristics of o. Then we relax o to o_r and augment o_r to o_a, such that any priority mapping satisfying o if it satisfies o_a. Finally, we calculate the probability lower-bound for random priorities satisfying o_a by exploiting its rooted-tree structure.

Consider the structure of o. The indexed events ordered by o comes from two sources: (1) $\mathrm{EI}(s^*)$, which is totally ordered in to_{Pri}; and (2) the delayed events added in Line 10 of Algorithm 3, which are only ordered (and must have lower priorities) to the events in the first part. The total number of events in (2) must be no more than $\mathrm{Confl}(s^*)$ according to Algorithm 3.

We relax any (a, b) in o if both (a, c) and (c, b) exist in o for any c - Pri satisfies o if and only if it satisfies the relaxed version of o. Continuously applying such reductions, the total order of events in part (1) becomes a chain. We thus call events in (1) *chain events*. For each of event in (2), its ordering can be reduced be lower than the lowest chain event in (1) among all the events it is originally lowered than in o. The final reduced ordering, denoted as o_r, forms a tree structure, containing the chain events and other events in (2), namely *leaf events*, attaching to the chain events.

Since $o \supseteq o_r$, each chain event has at most $|\mathcal{P}| - 1$ leaf events as children. Keeping

this as an invariant, we further "augment" o_r into o_a, such that any priority mapping satisfying must satisfy E_r (and also o). This is done by changing any (a, b) in o_r into (c, b), if (a, c) exists in the chain, and c does not have $|\mathcal{P}| - 1$ leaf children yet. After such augmentation, o_a still hold the invariant, and each leaf event in the tree is ordered as low as possible.

We now obtain the lower-bound of the theorem by calculating the probability of a random priority mapping satisfying o_a. To satisfy o_a in the shape of rooted-tree, the probability is calculated recursively. Given a (sub-)tree rooted at r with children $\{c_1, c_2, \ldots, c_k\}$, let $\mathrm{S}(r)$ denote the number of descendants of r, and $\mathrm{P}(r)$ denote the probability for a random priority mapping satisfying the (sub-)tree rooted at r. $\mathrm{P}(r)$ can be calculated as

$$\mathrm{P}(r) := \frac{\Pi_{c \in \{c_1, c_2, \ldots, c_k\}} \mathrm{P}(c)}{\mathrm{S}(r) + 1}$$

This is because that a mapping Pri satisfies the ordering of (sub-)tree rooted at r if and only if:

1. Pri independently satisfies each sub-tree of r's children, and
2. Pri makes the priority of r higher than its descendants of the tree.

Looking back at o_a, let w be the number of the leaf events in o_a. There are at most $l := \lceil \frac{w}{|\mathcal{P}|-1} \rceil$ events in the chain, starting from the bottom, that have children of leaf events. Assuming all of those l events have exactly $|\mathcal{P}| - 1$ leaf children, the

probability can be calculated as

$$
\begin{aligned}
&\Pr[\mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri}) \simeq s^* \mid \mathrm{Pri} \circ\!\!- to_{\mathrm{Pri}}] \\
&= |s^*|!\,\Pr[\mathrm{Pri} \circ\!\!- o_a] \\
&= |s^*|!\frac{1}{|\mathcal{P}|}\frac{1}{2\,|\mathcal{P}|}\cdots\frac{1}{l\,|\mathcal{P}|}\frac{1}{l\,|\mathcal{P}|+1}\frac{1}{l\,|\mathcal{P}|+2}\cdots\frac{1}{l\,|\mathcal{P}|+|s^*|-l} \\
&= \frac{1}{|\mathcal{P}|^l}\frac{|s^*|!}{l!}\frac{(a+l)!}{(a+|s^*|)!} && (\text{let } a := l\,|\mathcal{P}| - l) \\
&\geq \frac{1}{|\mathcal{P}|^l}\left(\frac{l\,|s^*|}{(l+a)(|s^*|+a)}\right)^{\frac{|s^*|-l}{2}} \\
&= \left(\frac{1}{|\mathcal{P}|}\right)^{|s^*|}\left(\frac{|\mathcal{P}|\,|s^*|}{|s^*|+a}\right)^{\frac{|s^*|-l}{2}}
\end{aligned}
$$

Finally, by Algorithm 3, $w \leq \mathrm{Confl}(s^*)$, thus the lower-bound of the theorem holds.

Since every conditional probability of the partitions is guaranteed by the same lower-bound. We can discharge the condition, yielding the probabilistic guarantee of POS*.

Corollary 3.7.

$$\Pr[\mathrm{Sample}_{\mathrm{POS}*}(\mathrm{Pri}) \simeq s^*] \geq \left(\frac{1}{|\mathcal{P}|}\right)^{|s^*|} r^u$$

Since $r \geq 1$ and $u \geq 0$, POS* is at least as good as random walk. We will compare this guarantee with the guarantees of other approaches in general settings in Section 3.4.1.

```
global int x = 0, y[n] = {0};
```

Thread 1	···	Thread n
`int i = 0;`		`int i = 0;`
`i = x++;`		`i = x++;`
`y[i] = 1;`		`y[i] = 1;`
`y[i] = 2;`		`y[i] = 2;`
`...`		`...`
`y[i] = m;`		`y[i] = m;`
`x--;`		`x--;`
`x++;`		`x++;`

Figure 3.2: An adversary program for random walk to have the worst error-detection guarantee

3.4.1 Probability Guarantees on General Programs

We now analyze how POS* performs on general programs compared to random walk and PCT. Consider a program with n processes and k total events. We introduce the conflict rate α of a program, defined as

$$\alpha := max_{s \in \mathcal{S}} \frac{\text{Confl}(s)}{|s|\,(|\mathcal{P}| - 1)}$$

We know that $0 \leq \alpha \leq 1$, and we consider α generally small, as it is common for a program to have substantial non-conflicting events, for example, accessing shared variables protected by locks, semaphores, and condition variables, etc.

Under this setting, for random walk, we can construct an adversary program with the worst case probability as $1/n^k$ for almost any α, by setting up one conflicting access at the beginning of each process and two at the end, as in Fig. 3.2, where all accesses to `y[i]` are non-racing. For PCT, since only the order of the αk events may affect the partial order, the number of preemptions needed for a partial order in the worst case becomes αk, and

thus the worst case probability bound is $\sim 1/k^{\alpha k}$.

For POS*, the maximum number of conflicts Confl is reduced to $\alpha(n-1)k$. Theorem 3.7 guarantees the probability lower bound as

$$\frac{1}{n^k}\left(\frac{n}{1+(n-1)\alpha}\right)^{(1-\alpha)k/2}$$

Thus, POS* advantages random walk when $\alpha < 1$ and degenerates to random walk when $\alpha = 1$. Also, POS* advantages PCT if $k > n$ (when $\alpha = 1$) or $k^{1/(1-\alpha)-1} > n^{1/(1-\alpha)}\sqrt{\alpha + 1/n - \alpha/n}$ (when $0 < \alpha < 1$). For example, when $n = 2$ and $\alpha = 1/2$, POS* advantages PCT if $k > 2\sqrt{3}$. In other words, in this case, POS* is better than PCT if there are at least four total events.

3.5 Implementation

The abstract algorithm of POS* requires a predetermined priority map, while the implementation can decide the event priorities on demand when new events appear, since they are independently random. The implementation of POS* is shown in Algorithm 4, where POS* schedules a program without explicitly constructing the schedule. *pri* represents the mutable random priorities of the next events for all processes. x is an object representing the current program state with the following interfaces:

- $x.\mathrm{Enabled}()$ returns the current set of enabled process.
- $x.\mathrm{Execute}(p)$ returns the resulting state object after scheduling process p in the state of x.

Algorithm 4 Testing a program with POS*

```
1:  procedure POS*(x)                                   ▷ x: the initial state of the program
2:      pri ← [ ]                                       ▷ Initially, no priority is assigned
3:      while x.Enabled() ≠ ∅ do
4:          p* ← τ                          ▷ Assume the special process id τ ∉ x.Enabled()
5:          for each p ∈ x.Enabled() do
6:              if p ∉ pri then
7:                  newPriority ← 𝒰(0, 1)
8:                  pri ← pri[p ↦ newPriority]
9:              end if
10:             if p* = τ ∨ pri(p*) < pri(p) then
11:                 p* ← p
12:             end if
13:         end for
14:         assert p* ≠ τ
15:         for each p ∈ x.Enabled() do                 ▷ Reset priorities
16:             if p = p* ∨ x.IsConflict(p, p*) then
17:                 pri ← pri \ {p}                     ▷ The priority will be reassigned in 6-9
18:             end if
19:         end for
20:         x ← x.Execute(p*)
21:     end while
22:     return x
23: end procedure
```

- $x.\mathrm{IsConflict}(p, p')$ returns if there is a conflict between the next events of p and p'.

Lines 15-19 of the algorithm are for the priority reassignment. If a conflict is detected during the scheduling, the priority of the delayed event in the conflict will be removed and then be reassigned at Lines 6-9.

3.6 Evaluation

To understand the performance of POS* comparing with other RCT approaches, we conducted two sets of experiments: We first measured how RCT approaches cover partial-orders in randomly generated micro benchmarks, then we studied how the approaches

find errors in macro benchmarks including real-world multi-threaded programs.

Our implementation and benchmarks are open-sourced online [75].

3.6.1 Micro Benchmark

In micro benchmarks, we randomly generated multi-threaded programs executing a small number of events without loops or conditions. Each program consists of n threads, and each thread executes m events accessing o objects. To make the schedule space tractable, we chose $n = m = o = 4$, resulting 16 total events. To simulate different object access patterns in real programs, we chose to randomly distribute events accessing different objects with the following configurations:

- Each object has respectively $\{4,4,4,4\}$ accessing events. (Uniform)
- Each object has respectively $\{2,2,6,6\}$ accessing events. (Skewed)

For each of the programs, we measured the coverage performance of each RCT approach by the minimum number of times that any partial order of the program is sampled, divided by the total number of runs of the approach.

The results are shown in Table 3.1. The benchmark columns show the characteristics of each generated program, including (1) the configuration used for generating the program; (2) the number of distinct partial orders in the program; (3) the maximum number of preemptions needed for covering all partial orders; and (4) the maximum number of conflicts in any partial order. On every program, we ran each RCT approach for 5×10^7 times (except for random walk, for which we calculated the exact probabilities). If a program was not fully covered by an algorithm within the run limit, the coverage is denoted

Table 3.1: Partial-order coverage on the micro benchmark programs. Columns under "benchmark" are program characteristics explained in Section 3.6.1. "$0(x)$" represents incomplete coverage. "Geo-mean" shows the geometric means of coverage, but conservatively accounts any incomplete coverage as $\frac{1}{5\times10^7}$

Benchmark				Coverage				
Conf.	PO. count	Max depth	Max confl.	RW	PCT	RAPOS	POS	POS*
Uniform	4478	6	19	2.65e-08	0(4390)	1.84e-06	0(4475)	7.94e-06
	7413	6	20	3.97e-08	0(7257)	3.00e-07	2.00e-08	5.62e-06
	1554	5	19	8.37e-08	0(1540)	1.78e-06	4.00e-08	8.54e-06
	6289	6	20	1.99e-08	0(6077)	1.34e-06	0(6288)	6.62e-06
	1416	6	21	1.88e-07	0(1364)	1.99e-05	1.80e-07	4.21e-05
Skewed	39078	7	27	5.89e-09	0(33074)	0(39044)	0(38857)	1.20e-07
	19706	7	24	4.97e-09	0(18570)	0(19703)	0(19634)	5.00e-07
	19512	6	27	2.35e-08	0(16749)	1.00e-07	0(19502)	1.36e-06
	8820	6	23	6.62e-09	0(8208)	1.00e-07	0(8816)	1.20e-06
	7548	7	25	1.32e-08	0(7438)	1.30e-06	2.00e-08	3.68e-06
Geo-mean				2.14e-08	2.00e-08	4.11e-07	2.67e-08	2.87e-06

as "$0(x)$", where x is the number of sampled partial orders. For PCT, we tweaked each case by adding a dummy event at the beginning of each thread, as otherwise PCT cannot delay the actual first event of each thread. We set the minimum parameters n, k, and d for PCT, such that it is able to sample any partial order in each case. The results show that POS* performed the best among all algorithms. For each algorithm, we calculated the overall performance as the geometric mean of the coverage. POS* overall performed $\sim 7.0\times$ better compared to other algorithms. ($\sim 134.1\times$ excluding RAPOS and POS)

In addition to the previous experiments, we measured the benefit of POS* to leverage fine-grained independence relations, such as those between read events. We generated another set of programs with the previous configurations, but half of the events are read-only. We compared POS* with its variant ignoring read-read independence, namely POS*-. The results are shown in Table 3.2. Overall, POS* performed $\sim 1.4\times$ as good as POS*-,

Table 3.2: Partial-order coverage on the micro benchmark programs with 50% read events. POS*- means POS* ignoring read-read independence.

Benchmark				Coverage					
Conf.	PO. count	Max depth	Max confl.	RW	PCT	RAPOS	POS	POS*-	POS*
Uniform	896	6	16	7.06e-08	0(883)	9.42e-06	2.00e-08	9.32e-06	1.41e-05
	1215	6	18	3.53e-08	0(1204)	8.70e-06	6.00e-08	1.22e-05	1.51e-05
	1571	7	17	8.83e-09	0(1523)	4.22e-06	0(1566)	7.66e-06	1.09e-05
	3079	6	15	1.99e-08	0(3064)	8.20e-07	1.20e-07	7.08e-06	7.68e-06
	1041	4	18	2.51e-07	0(1032)	3.05e-05	2.20e-06	3.32e-05	4.85e-05
Skewed	3867	6	19	6.62e-09	0(3733)	1.24e-06	8.00e-08	4.04e-06	4.24e-06
	1057	6	20	2.12e-07	0(1055)	4.68e-06	2.08e-06	2.79e-05	2.80e-05
	1919	6	20	2.09e-07	0(1917)	2.02e-06	3.80e-07	1.48e-05	1.48e-05
	11148	7	21	4.71e-08	0(10748)	4.00e-08	0(11128)	1.58e-06	3.02e-06
	4800	7	19	3.97e-08	0(4421)	5.00e-07	0(4778)	1.58e-06	4.80e-06
Geo-mean				4.77e-08	2.00e-08	2.14e-06	1.05e-07	7.82e-06	1.08e-05

$\sim 5.0\times$ better compared to other algorithms (and $\sim 226.4\times$ better compared to PCT and RW). This shows that POS* can benefit from fine-grained independence relations.

3.6.2 Macro Benchmark

To evaluate how POS* and other RCT approaches find errors in real world, we used SCTBench [63], a collection of concurrency bugs on multi-threaded programs. SCTBench collected 49 concurrency bugs from previous parallel workloads [11, 70] and concurrency testing/verification work [48, 14, 55, 20, 72]. SCTBench comes with a concurrency testing tool, Maple [73], which intercepts `pthread` primitives and shared memory accesses, as well as controls their interleaving. When a bug is triggered, it will be caught by Maple and reported back. We modified Maple to implement POS* and measure the conflict rates of the benchmark programs. The conflict rates are shown in Figure 3.3. The overall average of conflict rates is 0.2037, and majority of the benchmark programs have low conflict

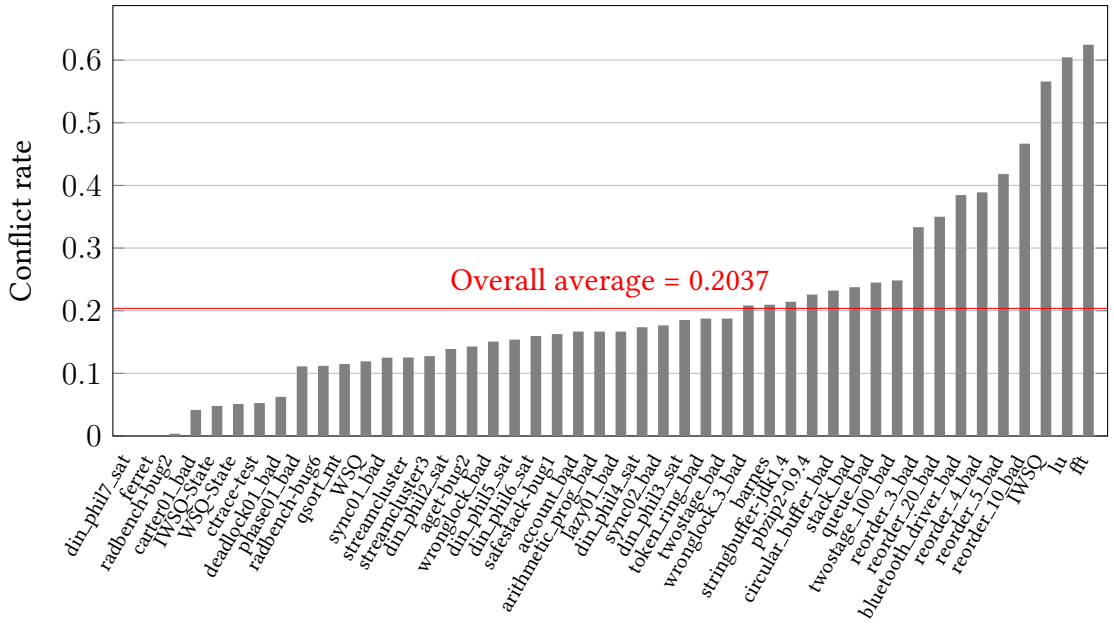

Figure 3.3: Conflict rates in macro benchmarks in ascending order over 1,000 trials in POS* scheduling. Rates are measured as $\frac{\max\{\#\text{ of conflicts}\}}{\max\{\#\text{ of events}\}\times(\max\{\#\text{ of threads}\}-1)}$. `fsbench_bad` is ignored since all its run crashes immediately.

rates that are less than 0.25.

Each sampling method was evaluated in SCTBench by the ratio of tries and hits of the bug in each case. For each case, we ran each sampling method on it until the number of tries reaches 10^4. We recorded the bug hit count h and the total runs count t, and calculated the ratio as h/t.

Two cases in SCTBench are not adopted: `parsec-2.0-streamcluster2` and `radbench-bug1`. Because neither of the algorithms can hit their bugs once, which conflicts with previous results. We strengthened the case `safestack-bug1` by internally repeating the case for 10^4 times (and shrunk the run limit to 500). This amortizes the per-run overhead of Maple, which could take up to a few seconds. We modified PCT to reset for every internal loop. We evaluated variants of PCT algorithms of PCT-d, rep-

resenting PCT with $d-1$ preemption points, to reduce the disadvantage of a sub-optimal d. The results are shown in Table 3.3. We ignore cases in which all algorithms can hit the bugs with more than half of their tries, filtering out trivial cases. The cases are sorted based on the minimum hit ratio across algorithms. The performance of each algorithm is aggregated by calculating the geometric mean of hit ratios on every case. The best hit ratio for each case is marked as blue.

The results of macro benchmark experiments can be highlighted as below:

- Overall, POS* performed the best in hitting bugs in SCTBench. The geometric mean of POS* is $\sim 2.6\times$ better than PCT and $\sim 4.7\times$ better than random walk. Because the buggy interleavings in each case are not necessarily the most difficult ones to sample, POS* may not perform overwhelmingly better than others, as in micro benchmarks.
- Among all 32 cases shown in the table, POS* performed the best among all algorithms in 20 cases, while PCT variants were the best in 10 cases and random walk was the best in three cases.
- POS* is able to hit all bugs in SCTBench, while all PCT variants missed one case within the limit (and one case with hit ratio of 0.0002), and random walk missed three cases (and one case with hit ratio of 0.0003).

3.7 Summary

We have presented POS, a new RCT framework, and POS*, a novel partial order method, to uniformly sample the partial orders of concurrent programs with significant guaran-

tees. POS*'s core algorithm is simple and lightweight: (1) assign a random priority to each event in a program; (2) repeatedly execute the event with the highest priority; and (3) after executing an event, reassign its racing events with random priorities. We have formally shown that POS* has an exponentially stronger probabilistic error-detection guarantee than existing randomized scheduling algorithms. Evaluation has shown that POS* is effective in covering the partial-order space of micro-benchmarks and finding concurrency bugs in real-world programs such as Firefox's JavaScript engine SpiderMonkey.

Table 3.3: Bug hit ratios on macro benchmark programs. "Geo-mean" shows the geometric means of hit ratios, but conservatively accounts any no-hit case as $\frac{1}{t}$, where t is the number of trials on the case.

Case	RW	PCT-2	PCT-3	PCT-4	PCT-5	PCT-20	POS*
01 stringbuffer-jdk1.4	0.0638	0.0000	0.0193	0.0420	0.0600	0.0332	0.0833
02 reorder_10_bad	0.0000	0.0007	0.0014	0.0017	0.0021	0.0000	0.0308
03 reorder_20_bad	0.0000	0.0015	0.0027	0.0040	0.0043	0.0021	0.1709
04 twostage_100_bad	0.0000	0.0000	0.0000	0.0002	0.0002	0.0000	0.0047
05 radbench-bug2	0.0003	0.0000	0.0010	0.0030	0.0045	0.0000	0.0418
06 safestack-bug1×10^4	0.0480	0.0000	0.0000	0.0000	0.0000	0.0000	0.2440
07 WSQ	0.0002	0.0484	0.0813	0.1054	0.1190	0.1444	0.0497
08 WSQ-State	0.0092	0.0003	0.0015	0.0017	0.0019	0.0146	0.0926
09 IWSQ-State	0.0643	0.0006	0.0040	0.0073	0.0121	0.0618	0.1380
10 IWSQ	0.0010	0.0461	0.0775	0.0984	0.1183	0.1205	0.0500
11 reorder_5_bad	0.0018	0.0061	0.0110	0.0122	0.0126	0.0089	0.0668
12 queue_bad	0.9999	0.0068	0.1415	0.2621	0.3511	0.6176	0.9999
13 reorder_4_bad	0.0074	0.0118	0.0206	0.0263	0.0294	0.0294	0.0795
14 qsort_mt	0.0097	0.0117	0.0239	0.0328	0.0398	0.0937	0.0958
15 reorder_3_bad	0.0246	0.0255	0.0457	0.0580	0.0660	0.0920	0.0997
16 wronglock_bad	0.3272	0.0351	0.0630	0.0942	0.1142	0.2508	0.4227
17 bluetooth_driver_bad	0.0628	0.0390	0.0597	0.0778	0.0791	0.1334	0.0847
18 radbench-bug6	0.3026	0.0461	0.0748	0.1011	0.1220	0.1435	0.2305
19 wronglock_3_bad	0.3095	0.0683	0.1137	0.1454	0.1741	0.2689	0.3625
20 twostage_bad	0.0806	0.1213	0.1959	0.2448	0.2804	0.2579	0.1212
21 deadlock01_bad	0.3668	0.0904	0.1714	0.2468	0.3160	0.8363	0.3315
22 account_bad	0.1173	0.2140	0.1929	0.1748	0.1628	0.1189	0.3367
23 token_ring_bad	0.1245	0.1367	0.1717	0.1923	0.2021	0.2171	0.1724
24 circular_buffer_bad	0.9159	0.1301	0.2888	0.4226	0.5180	0.7114	0.9369
25 carter01_bad	0.4706	0.1591	0.2974	0.4043	0.5007	0.9583	0.4999
26 ctrace-test	0.2380	0.2755	0.3342	0.3459	0.3453	0.2099	0.4680
27 pbzip2-0.9.4	0.3768	0.2321	0.2736	0.3048	0.3245	0.3609	0.6268
28 stack_bad	0.6051	0.2800	0.4060	0.4811	0.5365	0.7352	0.6210
29 lazy01_bad	0.6089	0.5386	0.5645	0.5906	0.6112	0.6887	0.3313
30 streamcluster3	0.3523	0.4970	0.5020	0.4979	0.5009	0.4849	0.4421
31 aget-bug2	0.4961	0.3993	0.4691	0.5036	0.5285	0.6117	0.9395
32 barnes	0.5180	0.5050	0.5049	0.5048	0.5052	0.5043	0.4846
Geo-mean	0.0380	0.0213	0.0459	0.0604	0.0692	0.0694	0.1795

Chapter 4

Effective Concurrency Testing For Distributed Systems

Towards effective RCT for real-world concurrent systems, we build Morpheus [74], an effective concurrency testing tool for distributed systems. Morpheus leverages POS to effectively manifest simple event orderings, which match real-world errors. To focus on protocol level errors, we target systems in Erlang, a high-level program language with first-class support in concurrent and distributed programming. Morpheus uses conflict analysis to avoid exploring unnecessary events always independent and further improve the effectiveness of exploration. We applied Morpheus in four popular real-world distributed systems, and found 11 new errors. All the errors are protocol-level and independently reproducible.

In the rest of the chapter, we first show a necessary Erlang background. Then we show an overview of Morpheus, including its work flow and techniques. Next we describe its implementation details, and show the errors found by Morpheus. We then present the evaluation of Morpheus. Finally we describe related work and conclude.

4.1 Erlang Background

Erlang is an actor-model based programming language with built-in concurrency and fault-tolerance. Its core primitives are listed in Tab. 4.1. In Erlang, a system is comprised

of *processes* (a.k.a. actors) distributed across *nodes*. Each node corresponds to an Erlang virtual machine with OS threads to schedule and execute Erlang processes in bytecode. Erlang processes are functional programs communicating with each other using messages without shared memory. A process sends messages asynchronously to the receiver process's mailbox, and synchronously receives messages from its own mailbox in the sending order. Erlang also maintains a shared namespace of processes running in the same node. This namespace allows a process to send messages to other processes referred by symbols. For brevity, this paper refers to an Erlang process simply as "process."

Erlang connects nodes in a distributed system via TCP/IP transport. A process communicates with remote processes in another node with the same primitives except with slightly more relaxed semantics. For example, messages sent to remote processes may be delayed instead of instantly delivered.

For fault-tolerance, Erlang detects, and by default, isolates any fault locally in its process. To explicitly handle faults, a process can, and often needs to, use the `monitor` and `link` primitives to get asynchronous notifications of faults from other processes. Note that a fault in Erlang does not necessarily indicate an error – Erlang allows a process to inject an artificial fault to another process, for example, to abort an on-going task in the recipient process.

For extensibility, Erlang supports interactions with non-Erlang code via calls to *native implemented functions (NIFs)* and messaging with *ports* (processes implemented in non-Erlang code). Furthermore, Erlang supports hot-swapping for overriding code without down-time, and loading dynamically generated code.

In sum, Erlang provide a lightweight, general set of concurrency primitives in the lan-

Table 4.1: Core concurrency primitives in Erlang.

Erlang primitives	Descriptions
`spawn(Node, F)`	Create a new process in node `Node` to execute function `F` with no argument, and return its pid.
`register(N, P)`	Register the process with pid `P` with the symbol of `N`.
`unregister(N)`	Unregister any process with the symbol of `N`
`whereis(N)`	Look up for the process with the name `N` and get its pid
`link(P)` `unlink(P)`	Link/unlink the current process with the process with pid `P`. Any fault will propagate from one process to its linked processes.
`monitor(process, P)`	Create a monitor on process with pid `P`, and return the monitor id. The monitor will send a message to the current process once `P` exits.
`demonitor(Mon)`	Cancel a monitor with id `Mon`.
`P ! M`	Send message `M` to the mailbox of the process with pid `P`.

`receive`
`  M1 when G1 ->`
`  % branch 1`
`  M2 when G2 ->`
`  % branch 2`
`  ...`
`after`
`  T ->`
`  % time-out`
`  % branch`
`end`
	Receive and handle a message in the mailbox if it matches any branch with its pattern (see language convention) and guard (expression that must be true). If no match is found after `T` milliseconds, take the time-out branch.

guage. Together with the high-level abstractions in the first-party Open Telecom Platform (OTP) libraries, Erlang serves as a solid foundation for building concurrent and distributed systems.

Language conventions. To understand the code examples in this paper, Erlang explicitly uses symbols beginning with capital letters or a underscore to refer to lexical variables

(e.g. `X` and `Proc`), and otherwise symbolic constants (e.g. `true` and `ok`). Erlang use pattern matching to match language object with syntactic patterns, binding data to any fresh variables in the patterns. For example, if `R` is a fresh variable, pattern `{reply, R}` matches the 2-tuple object `{reply, ok}`, and `R` will be assigned to symbol `ok` after the match. An Erlang project is comprised of modules, and modules functions. Code in one module calls function `Func` in another module `Mod` using syntax `Mod:Func(...)`.

4.2 Overview

We first describe the workflow of Morpheus (Section 4.2.1), then explain our techniques for effectively finding errors in real-world systems (Section 4.2.2 and Section 4.2.3), and finally discuss its limitations (Section 4.2.4).

4.2.1 Morpheus Workflow

Testing with Morpheus requires minimal effort, almost the same as testing with the standard facilities in Erlang. A high-level workflow of Morpheus is shown in Figure 4.1. Normally, a test case of a Erlang project is a function with a special name, so that the testing framework can discover and run it automatically. Morpheus takes the function as the entry, repeatedly executes it for multiple iterations to sample interleavings and discover errors. Each iteration begins with a Morpheus API call to execute the test function in a "sandbox", where Morpheus (1) isolates the test from external environment, (2) controls the execution of concurrent primitives, and (3) explores one of their possible interleavings.

Since Morpheus shares the same namespace as the test, it is crucial to isolate the test

to avoid unexpected inference between Morpheus and the test; otherwise errors caught by Morpheus may become not reproducible. Morpheus dynamically transforms modules, the unit of code deployment in Erlang, used in the test to isolate their namespace and references.

For execution control, Morpheus intercepts the invocations of communication primitives in the test. Since Erlang uses the actor model, where code runs in user-level processes (processes for short) in Erlang virtual machines, and communicates through messages, Morpheus mainly controls the messaging between the processes of the test. All primitives intercepted are replaced with our implementations, which (1) report the primitives to Morpheus as schedulable operations, (2) wait for Morpheus to resume the execution of the primitives, and (3) simulate the captured primitives or call the actual Erlang implementation of the primitives. Morpheus carefully handles each of the primitives to ensure it does not introduce behaviors impossible in the original executions without Morpheus.

With all schedulable operations gathered from the test, Morpheus schedules one operation at a time with a randomized scheduler to explore an interleaving of operations. Morpheus leverages POS to detect errors with high probabilities (Section 4.2.2). Each iteration ends when the test function terminates or an error occurs. Morpheus performs conflict analysis (Section 4.2.3) on the explored trace to identify conflicting operations that affected the testing result. With the analysis results, Morpheus predicts conflicting operations in next testing iterations, and focuses on reordering them to improve the probabilistic guarantees of randomized testing.

Finally, the testing result is reported to the developer, together with information for replaying the test with the same exact interleaving. The test succeeds if the entry function

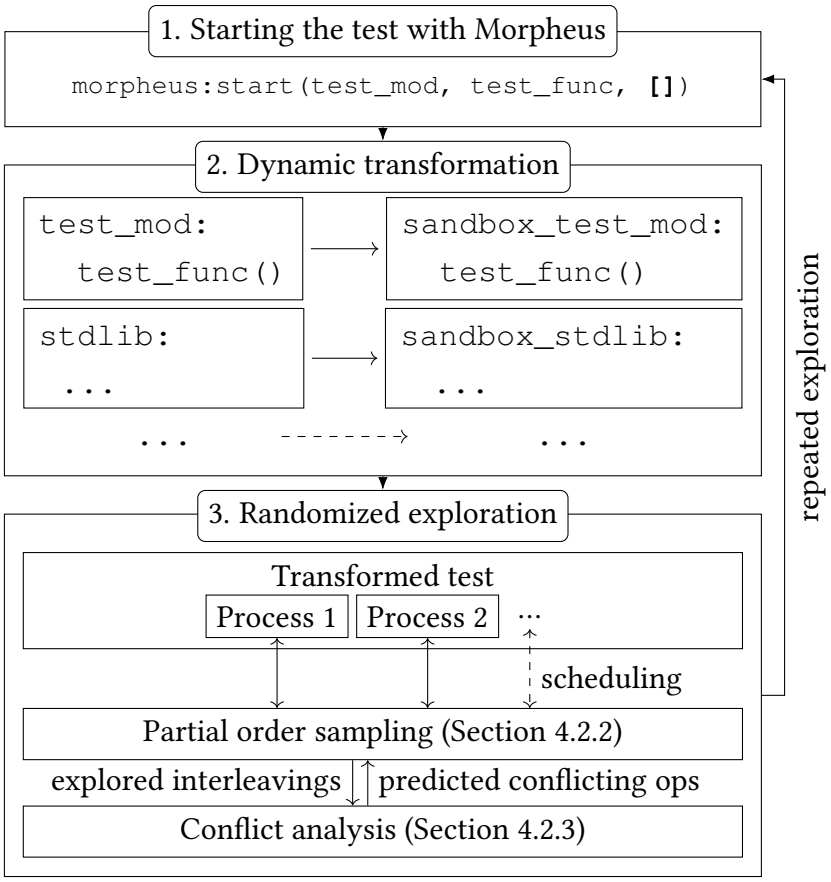

Figure 4.1: The workflow of Morpheus.

returns normally. The test fails if (1) the entry function exited with a fault, (2) the test calls into the Morpheus reporting API to report an error, (3) the test runs into a deadlock such that all processes are blocking on receiving messages, or (4) the virtual clock (Section 4.3) or the number of executed operations exceeds their user-configurable limits. A developer may run the test with Morpheus for many iterations to explore different interleavings and gain confidence.

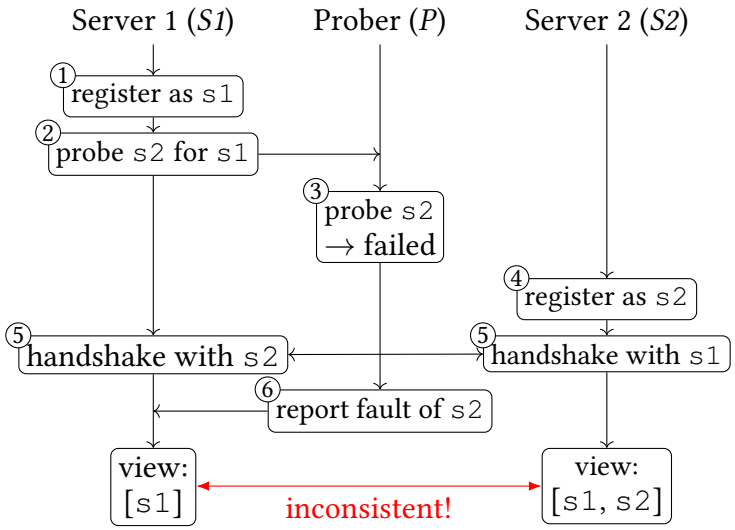

Figure 4.2: An inconsistent view error in `gen_leader`, a leader election protocol implemented in Erlang. "→ . . ." denotes the result of its preceding operation.

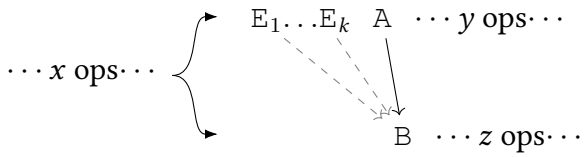

Figure 4.3: POS can sample the order of A→B with high probability that depends on the priorities assigned to A, B, and E_i, despite that there are numerous other x, y, and z operations.

4.2.2 Exploiting the Small Scope Hypothesis with POS

Previous study [38] showed that errors in real-world distributed systems have simple root causes, which sampling can hit with high probabilities. One must carefully choose a sampling strategy, since its bias may add up and degrade the probabilistic guarantee of error detection significantly even if the root causes of the error are simple.

A real-world example with a simple root cause. Figure 4.2 shows a simplified version of a real-world error we found in `gen_leader`, a leader election protocol implemented in Erlang and used by `gproc`, a feature-rich process registry in Erlang. This

error causes the system to get stuck and never handle any messages. It is triggered by an untimely fault message from Prober (*P*) to Server 1 (*S1*), which causes Server 1 and Server 2 (*S2*) to have different views of the alive servers even after a successful handshake.

In `gen_leader`, the leader election process of each node spawns a prober process to monitor other servers and handle any fault message from them. In bootstrapping, however, if the prober *P* of *S1* monitors *S2*, by querying the process registered as symbol `s2`, while *S2* has not registered as `s2` yet, *P* would report a fault message to *S1*. This fault message may be delayed due to, for example, OS scheduling. When later *S2* is up and handshakes with *S1*, both *S1* and *S2* enter the normal operational phase. Later *P* is resumed and sends the delayed fault message to *S1*, which will make *S1* believe that *S2* fails after the handshake. Depending on which server is elected as the leader, the delayed fault message may either make *S1* remove *S2* from its view of alive servers (if *S1* is the leader), or otherwise make *S1* roll back to election phase. In either case, further requests from `gproc` to `gen_leader` will not be handled as *S1* and *S2* disagree on the cluster membership and cannot recover. To detect this error, ④ must happen after the ①, ②, and ③, and ⑥ must happen after ④ and ⑤.

Redundancies in random walk. The major drawback of random walk is the exponential bias to delay one operation after another operation. In this example, random walk takes at most $(2/3)^k$ probability to delay operation ⑥ for k steps, the number of messages sent in the handshake ⑤. Real-world errors often require operations to delay such that random walk is unlikely to manifest due to the exponential probabilities.

Redundancies in PCT. As briefly described in Section 2.2.2, PCT [14] randomly selects a small number of operations to delay (preempt) by manipulating the priorities of processes right before the selected operations. Compared to random walk, PCT takes linear (to the number of operations) probability to delay a operation in contrast to the exponential probability. The major drawback of PCT is that it distributes the delayed operations among all operations in a test. Without precise knowledge of an error, however, a test often includes redundant operations irrelevant to the error, which "distracts" PCT from delaying the desired operations, and degrades its guarantees. In this example, PCT needs to (1) assign the priority of *S2* lower than priorities of *S1* and *P*, and (2) select ⑥ among all operations to delay. The original test of Figure 4.2, however, contains operations for system initialization before ①, and further operations after ⑥ for handling the actual `gproc` request. As a result, PCT has to guess the exact operation ⑥ to delay among all $\geq 6,000$ operations (as in Table 4.5) to manifest the error.

Finding errors in redundant tests with POS. Morpheus leverages the recent work of POS [76] to handle the redundancies in real-world tests. POS is simple: it assigns independently random priorities to operations, and always schedule the pending operation with the highest priority. Similar to PCT, POS also has the linear probability to delay an operation in contrast to the exponential probability of random walk. The key benefit we discovered about POS over PCT, is that the order of a subset of operations in a test only depends on priorities assigned to the operations in this subset. Thus POS is a good match for finding errors with simple root causes, even in redundant tests. Conceptually, consider the example in Figure 4.3. A prefix of x operations sets up concurrent processes

executing operations A and B, where B is preceded by $E_1, \ldots, E_k$. If an error is caused by the order of A→B, its scope would include operations E_i as well, because the scheduler must schedule E_i before B in order to schedule A before B, but the scheduler can schedule other $x + y + z$ operations arbitrarily without missing the error. To manifest the error, POS requires the random priority of B to only be lower than the random priorities of A and $E_1, \ldots, E_k$, which happens with the probability of $1/(2 + k)$, even with the presence of numerous other operations. Back to the real error of Figure 4.2, despite the operations of initialization and further user requests, POS would need only the priority of ④ lower than those of ①, ②, and ③, and the priority of ⑥ lower than those of ④ and ⑤,

A novel discovery in Morpheus is that, although the advanced version of the two previously proposed POS algorithms [76] performs 2∼3× better on multi-threaded programs, the basic version is actually (slightly) better in our evaluation. We believe the reasons are two-folds: the errors discovered in our evaluation do not reach the worst case of the basic POS. Moreover, the advanced POS suffers from the coarse-grained dependency tracking in Morpheus (and other tools such as Concuerror [17]) due to the complexity of message semantics in actor model compared to memory accesses. As pointed out in the original paper, in extreme cases, false dependencies could make advanced POS degenerate to random walk. Morpheus thus uses the basic POS as its default strategy.

4.2.3 Optimizing POS with Conflict Analysis

Although POS performs well to find errors in a small subset of operations, it still suffers from a form of redundant sampling as well. Consider the operation labeled ② in Figure 4.2.

When this operation is pending, POS assigns it a random priority, and the operation will be executed when it is the highest-priority pending operation. The effect is that POS samples the order of this operation with respect to other operations. However, this operation never conflicts with any other operations, and therefore can be immediately scheduled and any sampling of its order with respect to other operations is redundant. As shown in Section 4.5.3, we observed such non-conflicting operations as the majority of operations in our test cases, an order of magnitude more than the conflicting operations.

This shortcoming reflects a fundamental limitation of history-less randomized testing algorithms: whenever such an algorithm needs to schedule an operation, it has no idea what operations may occur in the future, therefore it has to assume that the operation may conflict with some future operations, and sample accordingly.

In contrast, partial-order reduction algorithms designed for exhaustive testing do not suffer from this problem. After the current iteration of testing is done, these algorithms examine the operations executed and detect which conflicting operations could have been executed in a different ordering. They deterministically backtrack only to these orderings (in the optimal case). However, modern partial-order reduction approaches (e.g. [25, 2]) require the testing to be depth-first, i.e. to backtrack an interleaving prefix only after all interleavings after the prefix are explored. Such requirement directly conflicts with randomized testing, therefore those approaches cannot be applied to POS.

Conflict analysis in Morpheus represents a novel design point in the randomized testing algorithm space. Compared to completely history-less randomized algorithms and deterministic algorithms with the full history of prior iterations, Morpheus keeps a concise summary of explored executions and use it to predict in future explorations. To do

that, Morpheus maps operations into compact signatures, such that operations with the same signature are likely to coincide on whether they are conflicting or not, and maintains a history table for each signature whether any operation with this signature has ever conflicted in previous explorations.

Specifically, after each test iteration, Morpheus runs a conflict detection algorithm on the explored execution and updates the history table. Morpheus detects conflicts by constructing the happens-before relation of the executed operations with vector clocks, and checking whether two operations accessing the same resource have concurrent vector clocks. If any conflicts are found, Morpheus records the signatures of the conflicting operations into the table. In next iterations, whenever Morpheus sees a new pending operation, it queries this table and immediately schedules the operation if no prior operations with the same signature ever conflicted.

An open design trade-off is the scheme of operation signatures. Intuitively, a signature scheme with more details would have better precision (i.e. less false positives) in the prediction of conflicting operations. Such scheme, however, would have larger overhead in time and space for bookkeeping. On the other hand, one could simply use the static code locations of the operations as the signature scheme, which may not help POS much due to its bad precision. Currently, Morpheus maps each operation into the signature of `{P, PC}`, where `P` is the process id of the operation and `PC` is its static location of code. In our evaluation, such scheme improved the precision from the naïve signature of static locations by up to 219.68% and improved the error-finding performance by up to 65.63%. Investigating more schemes for the trade-off of precision and overhead is left as our future work.

Beside false positives, conflict analysis could also have false negatives, where an operation is conflicting with future operations but Morpheus predicts otherwise. Such false negatives may involve slight unwanted bias for the current iteration, but it does not affect the probabilistic guarantees much because, after the current iteration, Morpheus will update the history table correctly to record that conflict involving this operation. In our experiments, we rarely saw false negatives after Morpheus populates the history table in a hundred of iterations.

4.2.4 Limitations

Morpheus focuses on testing concurrency and does not actively inject failures. Instead, Morpheus relies on test cases to manifest node failures, and explores any interleavings of the failures with normal system operations. In Erlang, it is straightforward to simulate node failures in tests by standard APIs, e.g., `erlang:disconnect_node`. We plan to leverage bounded and randomized approaches [71, 35, 44, 5, 10] to inject failures and find interesting errors. Morpheus does not support low-level failures such as packet loss.

Morpheus shares similar limitations with prior systematic or randomized testing tools. It relies on a test case for inputs to the tested program and application-specific invariants to check. (As aforementioned, it catches fail-stop errors and infinite loops automatically.) It supports limited ways of interacting with non-Erlang code (Section 4.3).

4.3 Implementation

We implemented Morpheus in ∼8500 lines of code in Erlang and a ∼50 lines of patch to the Erlang virtual machine in C++. We here describe some of its implementation details.

Instrumentation and isolation. As mentioned in Section 4.2.1, Morpheus transforms any module used in the test to (1) collect available operations, and (2) reliably control their interleaving without interference. Morpheus transforms a module by traversing the low-level syntax tree (of CoreErlang [15]) from the module binary, available as long as the module is compiled with debugging information. Morpheus traverses the syntax tree for each function definition, replaces all calls of concurrency primitives into calls of the Morpheus handler, where their functionalities are thoroughly simulated for both isolation and controlling.

To gather all schedulable operations, Morpheus needs to know when a primitive is available to execute, most primitives in Erlang are non-blocking, thus their transformed code simply reports to Morpheus. The only exception is receiving messages, which blocks until any message meets the receiving branches. Morpheus encapsulates the blocking semantics into a predicate (as in [18, 17]) that returns true if any branch is matched, as shown in Figure 4.4. Morpheus maintains its own virtual inbox to keep all pending messages in order, and uses the predicate to find the first matched messages in the inbox. If no message satisfies any conditions, it notifies Morpheus as blocking, and waits until new messages arrive or it times out.

To reliably control the interleaving, Morpheus always waits for all processes in the test to give back control before scheduling any operation. Sometimes an operation may have

Original receive

```
receive
  {reply, R}
    when R =/= error ->
      R
end
```

Extracted pattern function

```
PatFun =
fun (Msg) ->
  case Msg of
    {reply, R}
      when R =/= error ->
        true;
    _Other ->
        false
  end
end
```

Figure 4.4: `receive` pattern and its transformed matching function. The branch of the `receive` is taken only if the process receives a tuple with two elements, where the first element is symbol `reply`, and the second the element (assigned to `R`) is not symbol `error`.

concurrent side-effects. Morpheus enforces a deterministic order for such side-effects.

For isolation, Morpheus dynamically translates names referred in the test to avoid interference with Morpheus. Modules used in tests are dynamically transformed and loaded with new names with special prefixes (e.g. "`sandbox_`"). External references, e.g. files, are left uncontrolled for simplicity, as they usually can be configured in tests to avoid interference. If stricter isolation is required, a developer may use a container or virtual machine to execute the test.

Timing semantics. Real-world systems inevitably depend on real-time conditions to function correctly, and it is important to handle them in a desired way, otherwise we may spin with false errors impossible in practice. Currently Morpheus simulates the ideal semantic of timing by maintaining a virtual clock. All operations run infinitely fast and cost no virtual time, and time-outs happen in strict order of the their deadlines. All deadlines

of time-outs are maintained in a sorted queue. At any state, a timing operation is disabled unless its deadline is met. When there is no operation available but pending deadlines, Morpheus fast-forwards the virtual clock to the nearest deadline (similar to [71]), and triggers the time-outs whose deadlines are met.

To detect liveness errors where a system takes forever to finish a task, Morpheus limits the time of deadlines: Morpheus will not trigger any deadline after the limit. Thus any execution requiring more time than the limit will be reported as a deadlock. A few liveness errors lead to infinite loops without time-outs, for example, two processes keep sending/receiving messages with each other without handling test requests. Morpheus limits the number of operations to execute to detect such infinite loops. We believe that the timing semantics in Morpheus is a balanced trade-off between the program intention, testing effectiveness, and efficiency.

Simulating a distributed environment. Many tests of distributed systems require a distributed environment of multiple nodes. Instead of running physically distributed tests with Morpheus, error-prone and potentially having unwanted overhead, Morpheus simulates virtual nodes in a single physical node. Thanks to the unified process communication in Erlang, no extra transformation is needed for putting virtually remote processes in the same physical node. Morpheus needs to, however, properly translate remote name references. Since the Erlang compiler has special rules to prevents translation of some remote references, we combined source transformation with a minimal modification to the Erlang VM to do so.

For simplicity, Morpheus only simulates a fully connected cluster of nodes. Network

failure injection is not yet supported in Morpheus. A developer can inject node failures in her tests by shutting down all processes in the nodes, easy and covering all of our testing needs.

Non-Erlang code. Erlang systems commonly involve non-Erlang code for the sake of performance (e.g. cryptographic computation). Morpheus identifies all interactions with non-Erlang code during module transformation and runtime, and carefully handles them to avoid execution that is impossible without Morpheus. After each interaction with non-Erlang code, Morpheus waits for a small amount of time (50ms by default) to let the interaction stabilize.

For limitations, Morpheus assumes that external interactions are always non-blocking, stateless, and produce deterministic result. Morpheus ignores the operation dependencies produced by external interactions.

4.4 Errors Found

We applied Morpheus to four distributed systems in Erlang, and found 11 previously unknown errors. The target systems and errors found are summarized in Table 4.2. We targeted these systems because of their popularity and well-defined APIs. Most of the tests are written by us, since most original tests are non-concurrent or stress tests (except for `gproc`, where we used their test scenarios).

Creating tests for these systems were simple: we quickly studied their APIs by looking at their documents and original tests, identified intuitively error-prone scenarios, such as

Table 4.2: Summary of the distributed systems tested and errors founded by Morpheus

Name	Description	KLOC	Errors
`locks`	Lock manager	4.1	2
`gproc`	Process registry	7.3	3
`gen_leader`	Leader election	1.7	
`mnesia`	DBMS	27.3	2
`rabbitmq`	Message broker	60.7	4
`ra`	Replicated log	8.6	
Total		109.7	11

adding replicas to a node while concurrently shutting down the node, and implemented those scenarios using a few API invocations. We created the tests with no prior knowledge of the systems nor the errors. Each of these test scenarios contains 15-30 lines of code. Figure 4.5 shows one test we used on `locks`: With a few lines of setup, it creates concurrent processes to call `locks` APIs, and examines if the test can reach a normal termination.

Among all the 11 errors we found, three are confirmed by developers and all of them are reproduced afterward without Morpheus. We now describe the errors.

4.4.1 `locks`: a Decentralized Lock Manager

`locks` is a decentralized lock manager. It provides a transactional locking API for clients to request locks, detect deadlocks caused by circular waits, and surrender locks to avoid deadlocks. To acquire a lock `x`, a client sends a "request `x`" message to the lock server process in `locks`, The server in turn replies with the queue of the clients acquiring `x`, the head of which is the current holder of the lock.

Although the authors claimed that the core locking protocol of `locks` has been model

```
locks_1(Config) ->
  morpheus_guest_helper:bootstrap(),
  ok = application:start(locks),
  Me = self(),
  lists:foreach(fun (Id) ->
                  spawn(fun () ->
                    locks_1_client(Me, Id)
                  end)
                end, [1,2,3]),
  lists:foreach(fun (Id) ->
                  receive {Id, ok} -> ok end
                end, [1,2,3]),
  morpheus_guest:exit_with(success).

% Concurrent client processes
locks_1_client(CtrlProc, Id) ->
  {ok, Agt} = locks_agent:start(),
  LockOrder = case Id of
                1 -> [[1],[2],[3]];
                2 -> [[2],[3],[1]];
                3 -> [[3],[1],[2]]
              end,
  lists:foreach(fun (Lock) ->
                  locks:lock(Agt, Lock, write)
                end, LockOrder),
  locks:end_transaction(Agt),
  CtrlProc ! {Id, ok}.
```

Figure 4.5: The test scenario used to find `locks-1`. Three client processes concurrently acquire three locks in different orderings. For example, client 1 will lock in the ordering of `[1]`, `[2]`, and `[3]`.

checked despite minor differences in the implementation [68], we managed to find multiple errors in it, described in the following.

Prematurely resetting version (`locks-1`). In `locks`, the lock server and clients maintain the wait queues of locks in lock entries, tagged with monotonic version numbers

to identify out-dated entries in delayed messages. When a lock server *S* sees an available lock without any process waiting to acquire, it will remove the entry of the lock, effectively resetting its version. On the other hand, according to the locking protocol, a client may propagate its entries of some locks to another client that may or may not be waiting for the locks, in order to detect circular waits in a decentralized way. A concrete example is showed in Fig. 4.6, where three clients *A*, *B*, and *C* are acquiring locks `x` and `y` with server *S*.

Initially *A* and *B* acquire locks `x` and `y` separately. Then *A* and *C* attempt to acquire `y`, and `locks` notifies all three nodes that *B* is the current holder of `y`, and *A* and *C* are waiting in `y`'s queue in this order. Later, *B* requests `x` and detects a circular wait, so *B* surrenders `y`. It also propagates its entry of lock `x` to *C* in ① according to the protocol, even though *C* has not yet requested to acquire `x`. This design of the protocol is presumably for performance. After *A* and *B* exit, *S* removes the entry of lock `x` in ② because no one is requesting the lock, while the client *C* still has the now stale entry it received from *B*. When later *C* requests `x` from *S*, the request will never complete, because *S* will reply with a fresh entry of `x` "older" than *C*'s version. *C* will ignore the reply, and keep waiting for the "up-to-date" entry that will never arrive.

Atomicity violation in server probing (`locks-2`). In a cluster of nodes, when a client tries to acquire a global lock, the client needs to communicate with remote lock servers on other nodes. Since the remote lock servers may start later than the request, the client spawns a prober process in the remote node to wait for the server. The prober process first queries for any process registered as `server`. If not, the prober registers itself

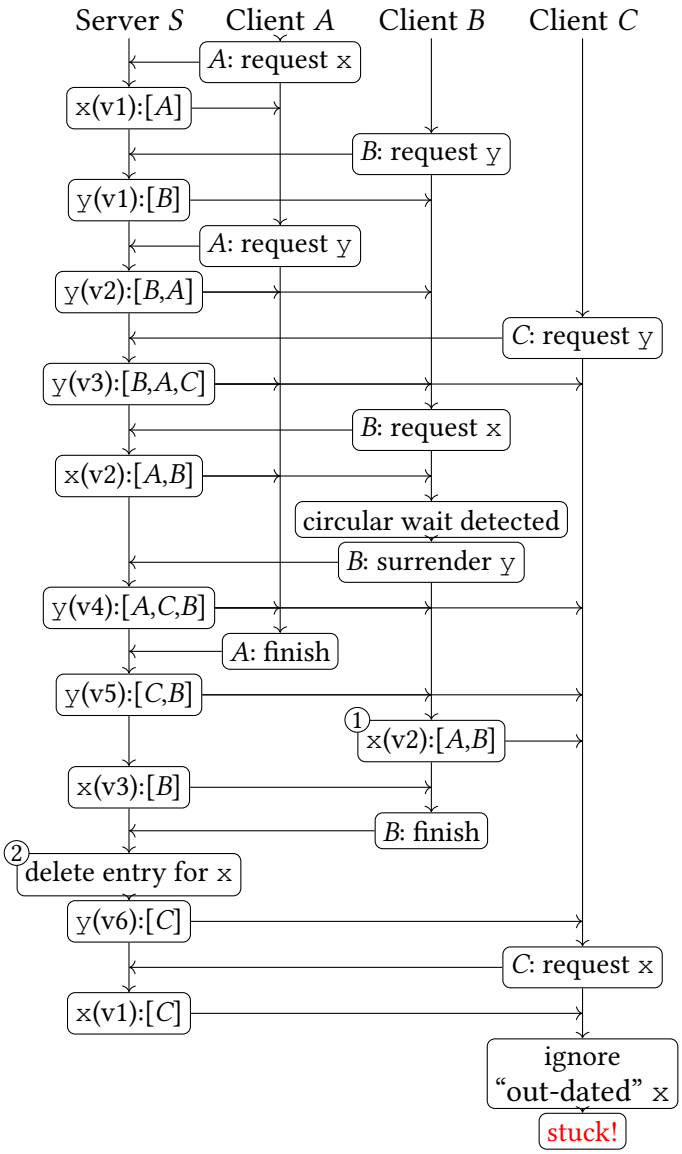

Figure 4.6: The deadlock caused by prematurely version resetting. `x` and `y` are names of locks.

as `watcher`, and the lock server will send notification to any process named `watcher` once the server starts. If the server starts after the probe query of `server`, but sends notification before the prober registers `watcher`, as shown in Figure 4.7, the notification will get lost and the prober will never detect the server, causing a deadlock.

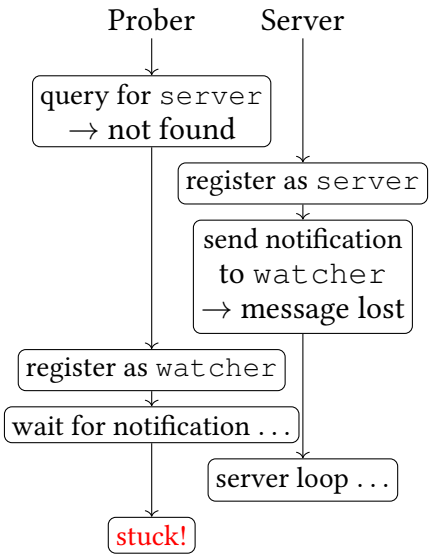

Figure 4.7: The atomicity violation in server probing. "→ . . ." denotes the result of its preceding operation.

4.4.2 `gproc` and `gen_leader`: an Extended Process Registry with Leader Election

`gproc` is a feature-rich process registry of Erlang subsuming the primitive process registry of Erlang. It also maintains the registry across a cluster of nodes on an leader elected by `gen_leader`, an implementation of leader election protocol. We tested `gproc` and `gen_leader` in two of our tests and one original test, which we describe in the following as well as presenting the errors found.

Delayed fault reports (`gproc-1`). We tested the bootstrapping procedure of `gproc` by setting up the distributed environment of three nodes, and then run a simple request of registering a process. We found that `gen_leader` could not handle some corner cases in bootstrapping, and may go in to an unrecoverable state where some nodes can never join the cluster. The simplified root cause of this error is showed in Figure 4.2 and described

in Section 4.2.

Reverted registration (`gproc-2`). We found an error of the `reg_other` API, which registers a remote process with a name. This error allows a successful registration of the remote process to be reverted, silently dropping all messages sent to the registered name. It happens as follows. Before the a node A joins a cluster (maintained by `gen_leader`), it has no knowledge of the cluster process registry because no node in the cluster sends it updates. However, a process in a node of the cluster can register A's process P with `reg_other`. Once A joins the cluster, it believes that it has the most up-to-date registry about its own processes, and removes the entry of P from the cluster registry. After that, all messages to the previously registered name will be silently dropped, breaking the contract of `reg_other`.

Lost requests with dying leader (`gproc-3`). It is important for leader election to tolerate node failures. We created a test where we first set up a cluster of three nodes, and then concurrently kill the leader node (by shutting down the `gen_leader` processes in the node) while issuing a `gproc` request on another node. Normally, `gproc` would use `gen_leader` to forward the request to the leader server. If the leader is dead, the cluster would go back to election and the request is buffered until the new election is done. If the request is sent after the leader is dead, but before the fault notification from the prober process, the request is lost and will never be replied to the client.

4.4.3 `mnesia`: a Distributed Database System

`mnesia` is a feature-rich database system that comes with the standard Erlang distribution[1]. It provides ACID transactions and replication across distributed nodes, and has been the standard storage solution for distributed applications in Erlang. We found two errors when `mnesia` handles multiple requests with simultaneous faults in a cluster.

Atomicity violation in initialization (`mnesia-1`). We tested the functionality of dynamically adding replicas of a table by requesting to replicate a table *T* from node *A* to node *B* while restarting the `mnesia` in *B*. We found a deadlock when the replication request happens in the initialization procedure of `mnesia` in *B*. When a user requests to add a replication of *T*, `mnesia` will execute a special schema transaction, which updates the list of replicas for *T*, in all nodes of the cluster. Concurrently, when *B* shuts down and restarts, the `mnesia` in *B* will execute multiple schema transactions to iteratively merge its schemas from other nodes. In our test, `mnesia` in *B* will create two schema transactions *X* and *Y* for iteratively merging schema, and transaction *Z* for adding the *B* to the list of replicas. The internal synchronization of `mnesia` makes sure that *X* always commits before *Z*, but when *Z* happens before *Y*, *Z* will grab the *T*'s schema lock in `mnesia`'s lock manager, and try to commit. Because *B* is still in the progress of merging schema (`mnesia` has a special flag for it), the commit of *Z* will be postponed in a queue until the schema merge is complete. When later *Y* starts in another process, it will try to grab the same table schema lock, but get blocked because *Z* is holding the lock and is not

[1]Erlang comes with the Open Telecom Platform (OTP) libraries of useful tools and middleware including `mnesia`

```
sync_schema_commit(_Tid, _Store, []) ->
  ok;
sync_schema_commit(Tid, Store, [Pid | Tail]) ->
  receive
    {?MODULE, _, {schema_commit, Tid, Pid}} ->
      ?ets_match_delete(Store, {waiting_for_commit_ack, node(Pid)}),
      sync_schema_commit(Tid, Store, Tail);
    % [MORPHEUS] Bug! Failure notification treated the same as commit
       ack
    {mnesia_down, Node} when Node == node(Pid) ->
      ?ets_match_delete(Store, {waiting_for_commit_ack, Node}),
      sync_schema_commit(Tid, Store, Tail)
  end.
```

Figure 4.8: The sync code for `mnesia` schema change, where the message of fault notification "{`mnesia_down, Node`}" is also treated as an acknowledgment of the commit.

committed yet. Thus Y and Z will form a circular wait and deadlock.

Reverted copy removal (`mnesia-2`). We created another test where we delete a node from the replicas of a table while restarting the node. We found an error that the schema change is not persistent in the resetting node, and the node reverts the change after the node is back. When `mnesia` runs the last phase of commit for the schema change, the node A initiating the change needs to waits for all other nodes to acknowledge their commits of the schema change. When another node B shuts down after entering this phase (i.e. after A sends the commit message to B), but before B persists the commit, B will send a fault notification to A. For some reason A will treat the fault notification as an acknowledgment of B (see Figure 4.8), and let the removal request succeed. Later when B is started again, it will replay its operation log without the last schema change, and bring the out-dated replica schema back.

4.4.4 `rabbitmq` and `ra`: a Message Broker with Raft Replication

`rabbitmq` is a sophisticated message broker service that implements high-level messaging protocols with rich semantics support and high performance. Its recent version provide high available "quorum queues" using `ra`, the implementation of Raft consensus protocol [53]. The Raft protocol maintains a consistent log replicated across a cluster. Whenever user requests come, the unique leader of the cluster, elected by the protocol, serializes the requests, appends them into a log, and safely replicates the log even in the presence of partial failures. We set up three test for `ra` that concurrently issues regular requests of enqueue/dequeue operations, leader elections, and multiple configuration changes concurrently, where we found three errors (`ra-{1,2,3}`). These errors are tricky such that the interleavings to surface them are highly complex. Once Morpheus produces these interleavings, however, diagnosing the root causes becomes easy. One of the errors happens when a `ra` server in a node is waiting for other servers to replicate the log entries missing in this server. When a concurrent election request comes, `ra` will buffer this request but forget the requester information by mistake. Later when the log replication is done, `ra` handles any buffered requests on the server and reply them. Since the election request was buffered without the source, `ra` will panic and crash unexpectedly. The other two errors involve `ra`'s incorrect assumption that, when reverting to an old view configuration, the node should always have the old view stored locally.

Besides `ra`, we also tested `mirrored_supervisor` module in `rabbitmq`, which maintains a replicated group of special "supervisors" processes to manage other processes across a cluster. We found an error (`ms-1`) in a test that concurrently adds and

removes supervisor processes from the group. The test initially sets up a group of supervisors *A* and *B*, adds a new supervisor process *C* to the group. To add *C* into the group, `mirrored_supervisor` acquires the current list of members (currently *A* and *B*), then it queries each of the members by sending messages. If we shutdown *B* after the list is retrieved but before the query to *B* is sent. The query will produce an error instead, which `mirrored_supervisor` does not expect, and will crash *C* unexpectedly.

4.5 Evaluation

Our evaluation of Morpheus focuses on the following research questions:

1. How does Morpheus's exploration algorithm, POS, compare with systematic testing with state-of-the-art partial order reduction techniques? Although POS has been shown effective in testing multi-threaded programs, it is unclear how much it helps testing distributed systems. (Section 4.5.1)
2. How effective does Morpheus find the errors with POS and conflict analysis, comparing with other randomized strategies? (Section 4.5.2)
3. How much can conflict analysis improve Morpheus? (Section 4.5.3)
4. How efficient is Morpheus to test real-world distributed systems? (Section 4.5.4)

We conducted all the experiments and studies in this paper on workstations with two Intel(R) Xeon(R) E-2640 CPUs and 64 GB of RAM, running Ubuntu GNU/Linux and our patched version of Erlang/OTP 20.3 and 21.3 (required by `ra`).

4.5.1 Comparison with Systematic Testing

To understand how randomized testing compares with systematic testing in distributed systems. We implemented randomized testing in Concuerror [17], a state-of-the-art systematic testing tool for Erlang equipped with advanced dynamic partial order reduction (DPOR) techniques [2, 9]. Our modification of Concurerror is small (∼500 LoC) as systematic testing shares most parts of runtime control and isolation with randomized testing, and only differs in scheduling strategies.

Concuerror is not suitable to test real-world systems for multiple design limitations. First, the test isolation of Concuerror is incomplete, and cannot properly handle interactions between the target systems and some stateful modules used by Concuerror. For example, most systems in Erlang use the stateful `application` module to set up themselves, but the module is not isolated by Concuerror. Secondly, Concuerror simplifies real-time conditions by allowing triggering time-outs immediately regardless of their deadlines. The simplification makes some common scenarios hard to test. For example, some tests need to wait for a delay to make the result stable after issuing the testing requests. We thus studied three simplified scenarios: the chain replication protocol checked by Concuerror in the previous work [8], the Erlang specification for a Cassandra error, and the "lock-1" error we studied. For a quick summary, POS outperformed systematic testing with DPOR in all cases we studied.

`crce`: chain replication protocol in `Corfu`. Previous work [8] demonstrated that Concuerror was able to find violations of linearizability on the specification of the chain replication protocol [57] as a part of Corfu distributed shared log system [45]. The authors

Table 4.3: Error-detection comparison on chain replication protocol. The "Systematic" column shows the number of iterations for systematic testing to find the error for each case. The "Random walk" and "POS" columns show the average number of iterations for the algorithms to find the error in each case.

Case	Systematic	Random walk	POS
`crce-2`	81	4.40	4.41
`crce-3`	119	1428.57	16.39

crafted four variants of the protocol specification, and they managed to find errors in two of them, namely `crce-2` and `crce-3`. We apply randomized testing on the same tests to compare with systematic testing. We profile the number of iterations for systematic testing to reach the first error. For randomized testing, we profile each of the algorithms by the average number of iterations needed for detecting an error over 10,000 iterations.

Table 4.3 shows the results. POS had the best numbers of iterations for finding the both errors. Random walk, while having similar performance on `crce-2`, performed badly on `crce-3` because of its unwanted bias against finding the error.

`C6023`: cassandra lightweight transactions. Inspired by previous case study on distributed system errors [38], we studied a sophisticated error [16] in Lightweight Transaction protocol of Cassandra [7]. We modeled the error in Erlang, and used Concuerror to check with systematic and randomized strategies. We performed 100,000 iterations for each of the strategies. Systematic testing and random walk could not find the error in all iterations, while POS was able to hit the error for 21 times.

The `locks-1` error. For all errors we have found with Morpheus, only `locks-1` can be tested in Concuerror without hitting its design limitation. The detailed error is de-

Table 4.4: Error-detection performance of different randomized testing algorithm. "+" means with conflict analysis. "**Mean**" row summarizes each of algorithms with the geometric means. "**Ratio to POS+**" row shows the ratio of overall performance compared to POS+. "**CA Improv.**" row shows the improvements of conflict analysis on the average performance for all algorithms. The summary of random walk is not available due to the missed errors.

Case	RW	RW+	PCT	PCT+	POS	POS+	POS*	POS*+
`locks-1`	0.0042	0.0312	0.0895	0.1164	0.1521	0.2087	0.1562	0.2239
`locks-2`	0.0210	0.0117	0.0022	0.0071	0.0073	0.0124	0.0103	0.0140
`gproc-1`	0	0.0001	0.0015	0.0018	0.0031	0.0106	0.0008	0.0023
`gproc-2`	0	0.0190	0.0496	0.0781	0.0605	0.1170	0.0416	0.0648
`gproc-3`	0	0.0002	0.0431	0.0385	0.0156	0.0450	0.0027	0.0094
`mnesia-1`	0	0.0001	0.0219	0.0223	0.0141	0.0168	0.0091	0.0124
`mnesia-2`	0	0.0008	0.0075	0.0078	0.0117	0.0253	0.0104	0.0254
`ms-1`	0	0.0061	0.3000	0.2833	0.1489	0.2420	0.1505	0.2478
`ra-1`	0	0	0.0076	0.0108	0.0070	0.0361	0.0062	0.0377
`ra-2`	0	0	0.0031	0.0032	0.0053	0.0127	0.0063	0.0131
`ra-3`	0	0.0004	0.0010	0.0007	0.0032	0.0092	0.0032	0.0100
Mean	N/A	N/A	0.0131	0.0157	0.0166	0.0369	0.0119	0.0265
Ratio to POS+	N/A	N/A	35.50%	42.55%	44.99%	100%	32.25%	71.82%
CA Improv.		N/A		19.85%		122.29%		122.70%

scribed in Section 4.4.1. Unfortunately, no DPOR technique implemented in Concuerror worked - they all timed out on planning the next interleavings to explore after the first iteration. We were not able to diagnose the issue, and it may indicate errors in their DPOR implementations. We instead ran the standard systematic testing with sleep-set method for 100,000 iterations, which covers 240 distinct partial orders, but no error had been found by systematic testing. For randomized testing approaches, random walk detected the error for only one time in 100,000 iterations. POS detected the error in 11,665 times in 100,000 iterations (i.e. $\sim$8.6 trials per error).

4.5.2 Morpheus Performance

For all the error we have studied in Section 4.4, we used Morpheus on the tests with different randomized testing algorithms, including random walk, PCT, and POS with and without conflict analysis (denoted with "+"). The original PCT algorithm was for multi-threaded programs, but it still applies here due to the instant message delivery in our concurrency model. For PCT, we set d, the number of preemptions, to five without the knowledge of the errors. We profiled each of the tests with 100 runs in POS to estimate the total number of operations for PCT. Besides of the basic POS, we also evaluated the advanced version, which not only assigns random priority to operations, but also re-assigns the priorities of pending operations when any conflicts are observed for the operations during the scheduling. We denote the advanced version as "POS*" hereafter.

For each combination of testing algorithms and tests, we performed 10,000 trials in 10 parallel tasks, so each task ran 1,000 trials. Each task starts with no history for conflict analysis. We collected for each error the hit-ratio, i.e., the number of trails that surfaced the error divided by the total number of trials. The results are shown in Table 4.4. Note that all three errors we found in `ra` appears in all our tests of `ra` with different probabilities. We aggregate the average results by the three errors. The hit-ratio of each algorithm is summarized in the "**Mean**" row using geometric mean. Random walk cannot detect 2 of 11 errors even with conflict analysis, thus Morpheus is infinitely better than random walk from the results.

The basic POS with conflict analysis performed the best in error-detection on average. It outperformed our baseline, random walk, and PCT, on most of the errors, and

the overall advantage is 181.70%. The results also show that, to our surprise, POS* (POS with priority reassignment) degrades the error-detection performance by 28% from POS in average, given the POS* worked better on multi-threaded programs in prior work [76]. Further experiments confirmed that POS* did sample more partial-orders than POS. These results reveal a key insight: tools consider two shared memory operations conflict only if they access the same exact location (and at least one is a write), but two message sends conflict if they target the same recipient process despite that the messages access disjoint parts in the recipient's state. Therefore, the partial-order relations defined on distributed systems tend to be overly conservative. Some of the false dependencies can be alleviated by fine-grained dependency tracking, proposed by Lukeman et al. [43], while semantic independence of operations is still hard to automatically discover.

4.5.3 Effect of Conflict Analysis

To understand how conflict analysis can reduce the useless operations for Morpheus to explore, we profiled Morpheus at two specific cases, `gproc-1` and `mnesia-2`, with the details of the total number of operations, real conflicts, false negatives, and false positives, under different signature schemes. The profiling setting is the same as Section 4.5.2. For each case, we compared the signatures of "`{P,PC}`" with the naive signatures of "`PC`", i.e. only the static locations. The data is showed in Table 4.5. Overall, the signatures of "`{P,PC}`" significantly reduced false positives from the static scheme, and have much better improvement for the error-finding performance (up to 342%). False negatives are extremely rare. False positives are at the same level of the real conflicts, and they com-

Table 4.5: Results of conflict analysis with different signature schemes on `gproc-1` and `mnesia-2`. "Hit-ratio" denotes the error-finding performance of Morpheus without and with conflict analysis. "FNs" denotes falsely ignored conflicting operations, and "FPs" denotes falsely explored non-conflicting operations.

Case	Operations	Conflicts	Hit-ratio
`gproc-1`	6593.39	325.59	0.0031
Scheme	FNs	FPs	
`PC`	0.09	1192.16	0.0064 (206%)
`{P,PC}`	0.54	526.27	0.0106 (342%)
`mnesia-2`	10018.80	628.25	0.0117
`PC`	0.18	4612.02	0.0174 (149%)
`{P,PC}`	1.13	1442.72	0.0253 (216%)

bined are a magnitude of smaller than the total number of operations for each of the tests.

The overall improvement of conflict analysis to Morpheus is shown in the "CA Improvement" row of Table 4.4. Conflict analysis improves the performance of POS by up to 415.71% in `ra-1`, and the average improvement is 122.29%.

4.5.4 The real-time performance of Morpheus

The overhead of Morpheus comes from three aspects: (1) transforming the modules on the fly, (2) controlling and executing the operations, and (3) analyzing the traces after each execution. We evaluated the real-time performance on two selected test cases, `gproc-1` and `mnesia-2`, over 100 iterations. We also measured the common time of a test by running an empty test without Morpheus. The results are shown in Figure 4.9. Overall, we observed nearly two orders of magnitude of overhead due to the heavy runtime manipulation, common in concurrency testing with full control. Module transformation took the majority of the overhead, which we believe can be amortized in future by reusing

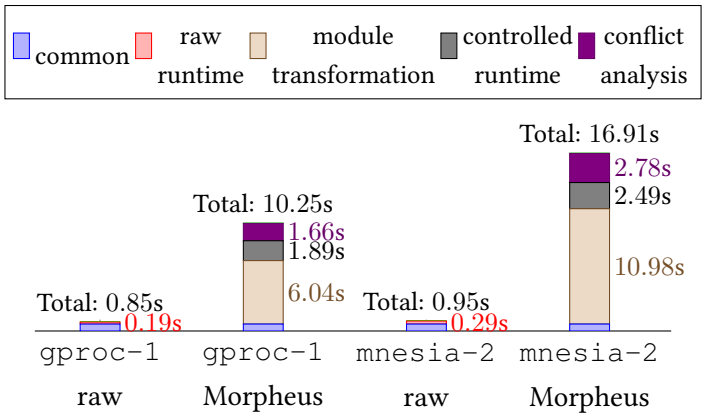

Figure 4.9: Runtime performance of Morpheus on `gproc-1` and `mnesia-2` over 100 iterations. The numbers on the right show the decomposition of runtime (except common).

the transformed modules. Comparing with PCT and random walk, the extra overhead of Morpheus is the conflict analysis, which contributes to 16.3% of the total runtime in average. Taking the extra overhead into account, Morpheus would still overall outperform PCT by 135.71% in real-time error-detection.

Comparing with Concuerror, we evaluated a mini-benchmark of sending messages in 1,000,000 round-trips, showing that Morpheus's runtime (94.02s) is 51.54% more efficient than Concuerror's (142.48s).

4.6 Related Work

Randomized concurrency testing. Burckhardt et al. proposed PCT [14] as the first randomized testing algorithm that leveraged the simple root causes of realistic concurrency errors, which has been later adapted into the context of general distributed systems [54] as PCPCT. Sen proposed RAPOS [58] that leverages partial order semantics for randomized testing as a heuristics, but it does not provide any probabilistic guarantees.

Jepsen [35, 44] tests real-world distributed systems by exploring their behaviors under random network partitions. For Erlang programs, QuickCheck/PULSE [18] provides randomized concurrency testing with a simple strategy that is akin to random walk. Morpheus leverages the Partial Order Sampling [76] in Morpheus to fully exploit the small scope hypothesis.

Systematic concurrency testing. A number of systematic testing tools have been proposed to apply model checking on implementations of concurrent systems, including CHESS [48], dBug [60], MoDist [71], DeMeter [30], SAMC [37], and FlyMC [43]. Some of them are for Erlang programs, such as McErlang [27] and Concuerror [17]. They explore the concurrency of a system and verify the absence of errors by exhaustively checking all possible interleavings. The major limitation of systematic testing is its ineffectiveness to handle the immense interleaving spaces to find simple errors. Partial order reduction techniques [29, 25, 2, 4, 9] alleviate this problem by skipping equivalent interleavings. For real-world systems, the interleaving spaces after reduction are often still intractable for systematic testing.

Program analysis for finding concurrency errors. There is a large body of work detecting concurrency error using static analysis [49, 65], dynamic analysis [24, 52, 42], and the combination of two [40, 41]. None of them alone can be effectively applied to real-world distributed systems with both coverage and precision.

Other approaches and language support for concurrency testing. Coverage-driven concurrency testing [73, 66] leverages relaxed coverage metrics to discover rarely

explored interleavings. Directed testing [59, 55] focuses on exploring specific types of interleavings to reveal targeted errors such as data races and atomicity violations. Some programming languages, for example, Mace [34] and P# [21] provide first-class support for building concurrent systems with high-level semantics, where our techniques can easily fit in.

4.7 Summary

We have presented Morpheus, an effective concurrency testing approach for real-world distributed systems in Erlang. Morpheus leverages Partial Order Sampling to fully exploit the small scope hypothesis and effectively find errors in complicated distributed systems. Morpheus targets the high-level concurrency for systems written in Erlang to focus on the protocol level errors, and introduces conflict analysis to further eliminate redundant explorations regarding the partial order semantics. Our evaluation showed that Morpheus is effective in finding errors in popular real-world systems in Erlang.

Chapter 5

Making Lock-free Data Structures Verifiable with Artificial Transactions

This chapter describes Txit, a system that simplifies the verification of lock-free data structures. Instead of directly verifying the implementations of data structures , Txit reduces their possible interleavings to a manageable size, such that one can exhaustively check/verify them with existing tools. Txit reduces the possible interleavings by grouping the code of the implementations into *artificial transactions*, each of which is guaranteed atomic. These transactions are added post facto after developers have written the code, hence we call them artificial. A tool now needs to verify only the interleavings of artificial transactions, an exponential reduction from the set of all shared memory access schedules. Once the data structure is deployed in production, Txit continues to enforce these transactions for correctness, and leverages hardware support to reduce the overhead of transactions. Txit thus automatically offers high assurance for legacy and new applications that use lock-free data structures, while retaining performance better than typical lock-based code.

Adding artificial transactions on execution can be quite dangerous, as in arbitrary code it may introduce deadlocks or live locks, demonstrated in prior study [12]. Fortunately, Txit does not suffer from this problem thanks to the obstruction-freedom of lock-free al-

gorithms, where a thread can always make progress with arbitrary transactions enforced.

A key challenge Txit faces is the tradeoff of performance vs verifiability (i.e., the number of schedules to verify). The granularity of artificial transactions determines the performance and verifiability of a lock-free data structure. Small transactions may not reduce the number of schedules adequately, while larger transactions yield fewer schedules, making the data structure much easier to verify, but increasing increase the probability of transaction conflicts, causing higher overhead for handling transaction aborts and retries. Thus, it is crucial for Txit to select a good plan to place transactions such that (1) all schedules of the data structure can be verified given a testing time budget and (2) the data structure with the inserted artificial transactions gives close to maximum performance under this testing budget. To tackle the challenge, we designed a heuristic search engine for empirically finding a high-performing transaction placement plan given a testing budget (Section 5.1).

We implemented Txit for C/C++ lock-free data structures. It leverages the LLVM compiler [36] to instrument programs and insert artificial transactions, the Pyevolve genetic programming engine [56] to search for an optimal transaction placement plan, the dBug model checker [60] to systematically check schedules of transactions, and TSX — the hardware transactional memory support readily available in the 4th generation Intel Core processors (codenamed "Haswell") [64] — to enforce artificial transactions (Section 5.2).

Evaluation on six popular lock-free data structures (Section 5.3) shows that:

1. Txit computes transaction placement plans such that the resultant data structures on the given test cases can be verified within several minutes by dBug.

2. The normalized execution time of Txit ranges from 1.55–4.30× using Haswell TSX.
3. According to our micro-benchmarking results, the overhead is primarily due to performance pathologies in Haswell TSX. For instance, transactional reads are (1) up to 1.63× slower than non-transactional ones and (2) are almost always *slower* than transactional writes.

Contributions. To the best of our knowledge, Txit is the first system that leverages transactional memory to aid verification of lock-free data structures. Our additional contributions include the idea of artificial transactions, the heuristic search engine for placing transactions, the results of verifying several popular lock-free data structures, and the discovery of the performance pathologies in the Haswell TSX support, along with our suggestions for improvements which we believe will benefit others wanting to use this feature.

5.1 Overview

In overview, we show a lock-free data structure example to illustrate the difficulty of writing and verifying such data structures; we show how Txit makes it easy to verify the example; and we describe the recommended usage of Txit.

5.1.1 An Example

Figure 5.1 shows a lock-free stack example and a test case exercising the stack. The `push` and `pop` operations appear correctly implemented because they use CAS to detect that

```
       void push(stack *s, element *e) {
         element *top;
         do {
push.1:    top = s->top;
push.2:    e->last = top;
push.3:  } while (CAS(&s->top, top, e) != top);
       }

       element *pop(stack *s) {
         element *top, *last;
         do {
pop.1:     top = s->top;
pop.2:     last = top->last;
pop.3:   } while (CAS(&s->top, top, last) != top);
         return top;
       }
```

(a)

`stack *s` is initialized as A→B→C→D

thread 1	thread 2
`element *x, *y;`	`element *x, *y;`
`t1.1: x = pop(s);`	`t2.1: x = pop(s);`
`t1.2: y = pop(s);`	`t2.2: y = pop(s);`
`t1.3: free(y)`	`t2.3: push(s, x);`
	`t2.4: free(y)`

(b)

Figure 5.1: A lock-free stack (a) and a failure-causing test case (b).

the stack top is changed and retry accordingly. However, the code actually suffers from a subtle bug that causes the same element to be popped twice. Consider this scenario: after thread 1 gets the stack top and sets `last` to point to element B, thread 2 pops two elements and the pushes back element A. Now, when thread 1 runs the CAS instruction to detect conflicts, the stack top is still A, so the CAS succeeds but incorrectly sets the stack top to point to B. When thread 1 continues to pop the next element, it gets B, causing a double free. This bug is a classic ABA bug [1, 46], which is common in lock-free data structure implementations.

Finding this bug is hard because even the simple test case shown in figure 5.1.(b) has an enormous number of schedules, estimated by DBUG to be 9×10^{22}. After utilizing state-

of-the-art state space reduction technique, dynamic partial order reduction (DPOR) [25], the number of schedules is still estimated to be 2×10^7.

5.1.2 Txit Work Flow

We describe how to make the stack example easy to verify with Txit. To reduce the set of schedules, Txit inserts artificial transactions. It starts by transforming each operation of the stack into a transaction, maximizing verifiability. Concretely, Txit makes `push` and `pop` transactions, reducing the number of schedules down to only 10, eliminating the ABA bug in `pop`.

This baseline transaction placement plan may incur high overhead, so Txit performs a search to find a good transaction placement plan. It guides the search using an evaluation function that (1) quantifies performance by measuring the execution time of the test case and (2) ensures that the estimated number of schedules is smaller than the testing budget ("estimated" because counting the precise number requires fully exploring the schedules).

After Txit finds an optimal plan with good performance and a verifiable set of schedules, it outputs a new stack implementation with transactions inserted. Developers then run their favorite tools to verify the correctness of this implementation, and deploy the implementation in production environments, where transactions are running under hardware support with minimal overhead.

5.1.3 Recommended Usage

While in principle developers can use Txit in any development stage, we recommend a specific stage—after traditional testing, but before deployment—because we believe Txit is the most useful in this stage. During active development, the code frequently changes, and for each version of the code, Txit may produce a different transaction placement plan, so its usefulness is limited. Txit provides high assurance by reducing the set of schedules, and the removed schedules may effectively hide bugs. Thus, developers should do testing/verification as usual without Txit to find as many bugs as possible, and turn on Txit as a final step to get high assurance in production environments.

5.2 Implementation

In this section, we show how we utilize the architecture support and model checker to build Txit, shrinking the schedule space of lock-free algorithms using artificial transactions. We will first describe an overall architecture, then explain each component in detail.

5.2.1 Architecture

Txit takes a lock-free data structure and a comprehensive[1] test case for input. The lock-free data structure is given as a library with exported interface functions. The whole workflow is shown in Figure 5.2.

[1]We rely on experts with domain knowledge to provide test cases with good coverage for testing desired properties.

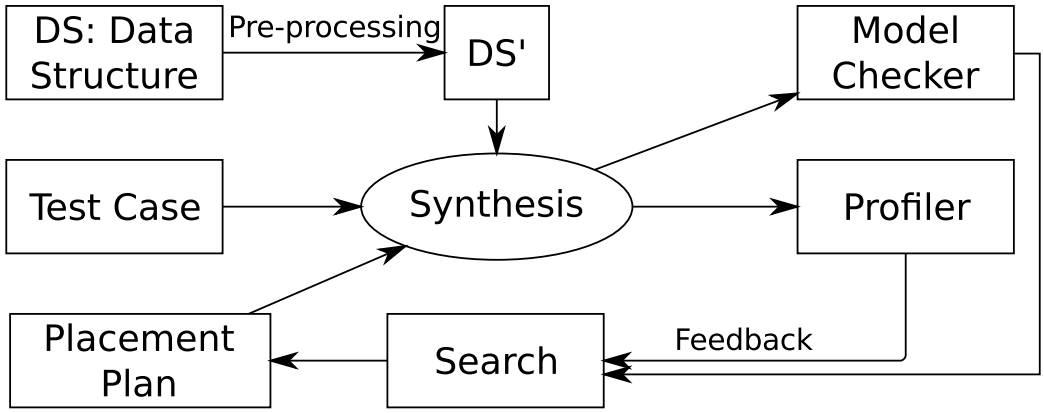

Figure 5.2: The architecture of Txit. Placement plans are synthesized into a proposed program, which is profiled and checked, and which feeds into new placement plans.

Taking the data structure and test case as input, Txit gradually improves the current placement plan by synthesizing checkable and runnable programs, feeding them into the model checker and profiler, and using their reports as feedback. After a number of iterations, the system outputs the best placement plan it has found.

Note that the system does not verify the source program. It finds a transaction placement plan to reduce the set of schedules needed to reason about for further verification. The goal is to get the best performance with a set of schedules that is still verifiable within the given budget.

5.2.2 Pre-processing

Given the data structure as a code library, it needs to be pre-processed before synthesizing and profiling. All the pre-processing procedures are done using LLVM IR transformations.

We first try to *flatten* the library so that most function calls are inlined, which expands the control flow and data flow, so they become cleaner and easy to deal with statically. Due to theoretical and resources constraints on the compiler, not all function calls can

be expanded; in which case we leave them untouched. Note that this may incur more pressure on the L1 code cache. Since the lock-free data structures are relatively small, in our experiments we did not observe any slowdown due to the extra pressure.

After the transformation, we identify all memory accesses in the library, and intercept them by appending hook functions. This is much more heavyweight than the original memory accesses. We only enable this instrumentation during model checking.

5.2.3 Model Checker Enhancement

To perform model checking on the program under a given transaction placement, we leverage the model checker DBUG. DBUG intercepts synchronization functions (such as `pthread_mutex_lock`) to track and control the scheduling. But it is not aware of any instruction level memory accesses, nor transactions. We modified DBUG to support the semantics of transactions, by extending it with two synchronization primitives: `TXBegin()` and `TXEnd(read_set, write_set)`. `TXBegin` starts a transaction and `TXEnd` commits the transaction with read/write sets collected by instrumentation. DBUG simulates the schedule by enforcing the total order of all synchronizations. During the checking `TXBegin` suspends all other threads so that only one transaction can be active for a given time.

5.2.4 Intel Haswell TSX Runtime

To utilize the hardware TSX support, we wrapped the instruction level interface into routines, `tx_begin` and `tx_end`, to start and commit a transaction. The tricky detail is

how TSX deals with conflicts. TSX detects conflicts by monitoring local cache lines involved in the ongoing transactions. Once such a cache line is invalidated or degraded (e.g. from "exclusive" state to "shared" state), TSX will discover the conflict against concurrent transactions in other threads, and abort the local transaction to resolve the conflict. This makes the local transaction exit transactional mode, and jump to a fallback branch prepared before the transaction, which does not guarantee the progress of transactions. One could let failed transactions retry until success, which could simply lead to a live lock. The optimization guidelines of TSX [31] require the programs to always provide a non-transactional fallback path for each transaction and must not simply let the transaction retry. Since the lock-free code is not aware of transaction enforcement, we need to provide our own fallback path. We used an exponential back-off strategy to resolve the contention in a decentralized way, which we believe is more scalable than global critical sections as the number of conflicting objects increases.

There are some situations where transaction aborts are not because of conflicts. For example, accessing an unmapped page will cause a page fault and cause an abort unless in a non-transactional fallback. TSX provides a value in `EAX` to identify the reason of an abort. Txit runtime will detect such situations and fallback to a global critical section.

Txit also leverages dBug to evaluate the verifiability of a placement plan. This is done by extracting the schedule space estimation from dBug. dBug computes this by dividing total number of states explored by the sum of probability of each explored state.

5.2.5 Genetic Search

Txit performs a genetic search on transaction placement plans, which are represented as boolean vectors. Each element in a plan vector denotes whether or not to insert a transaction bounary at a given location. We only consider boundaries before and after memory access instructions, so the size of a plan vector is fixed to twice the number of memory access instructions in the given program. The genetic search maintains a population of placement plans, and updates the population by randomly picking existing plans for crossover and mutation. Initially, the population is generated with random boolean vectors with the fixed size.

5.3 Evaluation

Our evaluation focuses on three research questions:

1. Can artificial transactions reduce the number of schedules effectively?
2. Can Txit find transaction-placement plans that offer good verifiability and performance?
3. What is the overhead of Txit with current hardware transactional memory? Where does the overhead come from?

5.3.1 Evaluation Setup

Our evaluation is performed on a work station with 16 GB of memory and Intel(R) Core(TM) i7-4770. This CPU has four cores and up to two hyper-threads per core, but

Library	Data Structures Selected
boost::lockfree [13]	stack (BLFS), queue (BLFQ)
folly (Facebook) [26]	producer-consumer-queue (FPCQ)
liblfds [39]	stack (LFDSS), queue (LFDSQ)
nbds [50]	skiplist (NBDSSL)

Table 5.1: Evaluated lock-free data structures.

we disabled hyper-threading per recommendation of the Intel manual. We locked the CPU frequency to 3 GHz to avoid inaccurate measuring caused by frequency scaling. The workstation runs Debian with Linux 3.11.

We selected 6popular open source implementations of lock-free data structures, shown in Table 5.1.

Folly from Facebook contains two data structures claimed to be lock-free, including FPCQ and atomic hash array. However, Txit detects a deadlock after adding transactions to the atomic hash array. It turns out this data structure is actually not lock-free: it spin-waits on hash array slots, violating lock-freedom. We thus leave this data structure out in our evaluation.

We used the following test cases to exercise the data structures. For general stacks and queues, the test cases spawn three threads, where each thread pushes two elements onto the stack/queue and then pops them out. For single consumer/producer queue (FPCQ), the test case spawns a producer thread and a consumer thread, where each thread pushes/pops the queue six times. For skip-list, the test case spawns three threads, where thread $i \in \{0, 1, 2\}$ inserts an element with keys $\{i, i + 3, i + 6\}$ into the skip-list, making the threads fully interleave in the insertion process.

Figure 5.3: Number of schedules (in log scale) vs transaction size. The horizontal line is at 10^4, and any result below this line is exact.

5.3.2 Reduction on the Number of Schedules

Here we evaluate how artificial transactions reduce the number of schedules with unified transactions size. Specifically, we show how the number of schedules grows (or shrinks) as the transaction size varies. For each test, we group every n shared memory accesses into a transaction and use DBUG to determine the number of schedules. Here $s_{\text{est}}(1)$ indicates that each instruction is in its own transaction, and $s_{\text{est}}(\infty)$ indicates that all instructions within an operation of the lock-free data structure are in one transaction. We run DBUG for up to 10^4 iterations to estimate the number of schedules; any S_{est} smaller than 10^4 is a exact result. Figure 5.3 shows the result with y-axis in log scale, demonstrating huge reduction in the number of schedules as the transaction size grows.

5.3.3 Performance and Verifiability Tradeoff Results

In this section, we evaluate how well TXIT makes the performance and verifiability tradeoffs. Given different testing budgets s_{budget} expressed as the number of schedules that developers can afford to test, TXIT's heuristic search engine explores the possible transaction placement plans, evaluates the plans according to the resulting state space and performance reports, and evolves them using genetic algorithms. In a few cases, increasing the testing budget did not improve performance because a solution for a smaller budget is faster.

To better understand these cases, we adjusted the evaluation criteria slightly to di-

Budget	Baseline	2×10^3	2×10^4	2×10^5	2×10^6
FPCQ	2.458	N/A	2.363	2.601	1.553
NBDSSL	2.166	2.142	1.952	2.355	1.935
BLFQ	3.649	3.628	3.232	3.321	3.063
BLFS	3.747	3.521	3.133	3.308	3.126
LFDSQ	3.184	4.304	2.705	2.569	2.565
LFDSS	2.481	2.776	1.956	2.916	2.047

Table 5.2: Normalized execution time of the test cases with transactions over without (smaller is better). The baseline column shows the normalized execution time at the starting point of the search for each data structure when every operation is made a transaction.

rect the search toward a solution whose number of schedules is close to the budget. We used the following genetic search parameters: population size of 70 and 80 generations, resulting in 5600 iterations for each data structure and testing budget.

Table 5.2 shows the results. Each cell shows the normalized overhead of running a test case with transactions over without. The baseline column shows the normalized overhead for the starting point of the search, i.e., when each operation of the data structure becomes one transaction and the verifiability is maximized. Txit did not find a valid placement plan for FPCQ when $s_{\text{budget}} = 2 \times 10^3$ because that baseline solution already has more schedules than the budget.

Running the parallel test cases on these placement plans shows that our searching system is able to reduce the running time of the baseline enforcement, to between 89.3% and 63.2%. Compared to the original performance without transactions, the numbers show the factors from 155.3% to 430.4% across all placement plans from search.

To understand what costs contributed to the overhead, we rerun these plans but using test cases that spawn only one thread; results from these experiments measure the

Budget	Baseline	2×10^3	2×10^4	2×10^5	2×10^6
FPCQ	1.273	N/A	1.539	1.530	1.863
NBDSSL	1.155	1.159	1.193	1.363	1.222
BLFQ	1.291	1.369	1.326	1.409	1.443
BLFS	1.299	1.399	1.404	1.476	1.378
LFDSQ	1.080	1.380	1.491	1.440	1.584
LFDSS	1.090	1.175	1.253	1.405	1.392

Table 5.3: Normalized execution time of single-threaded test cases with transactions over without (smaller is better).

operational cost of the transactions without any conflicts.

Table 5.3 shows the results. Operational cost is quite high, ranging from 17.5% to 53.9%. Even if we insert only one transaction for each operations, as in the baseline, the operational cost is still observable (8% to 29.9%). The implication is that the operational cost of Haswell transactional memory is relatively large compared to the size of the transactions we insert. We examine this cost in greater detail below.

5.3.4 Understanding Haswell TSX overhead

We have seen that transactions in the current Haswell implementation carry significant overhead even if there is no conflict. Since the operations of the evaluated lock-free data structures take hundreds to thousands of cycles, we wrote a microbenchmark at the same scale to study the behavior of Haswell transactions.

Microbenchmark. We designed the microbenchmark to be memory intensive to resemble the behavior of lock-free data structures. The benchmark consists of auto-generated functions that access a given work space of memory that fits in L1 data cache

```
  1 MOVL %eax, 0x000(%rdi)  ---- start of the first pass
    ...
256 MOVL %eax, 0x1F8(%rdi)  ---- end of the first pass
257 MOVL %eax, 0x000(%rdi)  ---- repeat (second pass)
    ...
```

Figure 5.4: Microbenchmark for evaluating TSX overhead.

(2 KB in our experiments). We first generate a instruction sequence that accesses the work space with a given stride, then we generate the benchmark function by repeating the sequence up to a given total instruction length. The structure of the microbenchmark is shown in Figure 5.4.

Cache Settings To compare the performance under different cache settings, we prepare a separate memory space to fill out the L1 cache, so that we can observe the overhead comparison of cold and warm cache. We place data in L1 for the warm case and in L2 for the cold case (to avoid L3 or main memory delay). We also run tests where the relevant L1 cache lines are all initially in a modified state, and where they are in an exclusive state. There is a significant flushing cost (compared to cache hit without eviction) when evicting modified cache lines, since TSX needs to flush dirty cache lines even for cache hits. Taking all combinations into account, we have four cases: WarmModified (WM), WarmExclusive (WE), ColdModified (CM) and ColdExclusive (CE).

We measured the TSX characteristics in each case. According to the structure of the microbenchmark, there ought to be two phases: the first phase touches all cache line in the read/write set, incurring extra cost; after that all data are warmed in the cache, ameliorating the performance. In the experiments, the first phase lasts from line `1` to `256`. We calculate the per-instruction overhead for the first phase. For comparison, we con-

Cache Setting		CE	CM	WE	WM
TX Mode?	Mem Op				
Yes	Read	1.45	1.44	1.13	1.42
No	Read	1.10	1.13	0.54	0.54
Yes	Write	1.20	1.17	1.07	1.70
No	Write	1.17	1.17	1.07	1.07

Table 5.4: Comparison of load and store instruction cycles under different conditions.

ducted the same experiments under non-transaction mode. Table 5.4 shows the results. Transactional load and store instructions are quite costly in TSX in almost all cases:

- Transactional reads are 27% to 163% slower than reads in normal mode, especially under WarmModified condition.
- Under WarmModified condition, transactional writes also have significant overhead (59%).
- Additionally, TSX shows a 70-cycle average overhead per transaction in all experiments. This may come from the memory barrier effect on the boundaries of transactions.

Summary of TSX overhead Based on our microbenchmark results, we believe the performance pathologies of TSX come from the following sources:

- **Cache Impact.** To isolate the memory accesses of transactions, TSX keeps the write set in local cache, which means the original value must be saved somewhere else. When instructions in a transactional region access a local cache line which has been previously modified but not touched in the current transaction, the CPU needs to backup the value to lower level cache in order to save the original value. This is

similar to evicting dirty cache lines, but here the CPU is "cleaning" the dirty cache line. This only happens in transactional mode, and make a L1 cache hit effectively a write back to L2. We suggest that TSX has a *specialized write back buffer* to resolve this issue.

- **Memory Barrier.** According to Intel's manual [32], a successfully executed transaction has the same memory ordering semantic as "lock" prefixed instructions. When TSX is used heavily in fine-grained, memory throughput will be reduced because of the effect of memory operation serializing.
- **Time Window Inflation.** Because of the hardware overhead, the time window of a transactional region is greatly inflated. This makes the transaction more likely to abort because of conflicts, and thus waste more time on retrying. Under the best effort protocol of TSX, which always fails the transaction that accesses the data first, time window inflation makes the transaction much harder to proceed under contention.

We expect these issues to be fixed in the next generation of TSX.

5.4 Summary

We have presented Txit, a system for making it easy to verify lock-free data structures, one of the most scalable and efficient among all classes of parallel programming abstractions. The key idea is to insert artificial transactions to reduce the set of schedules, while enforcing the transactions in production environment for correctness. Leveraging recent advances in hardware transactional memory support – specifically Intel Haswell TSX –

Txit achieves acceptable performance. The granularity of artificial transactions affects the performance and verifiability: larger transactions yield fewer schedules and better verifiability, but reduce performance because of the increased probability of transaction aborts. In our evaluation, we have demonstrated that Txit reduces the set of schedules enough that tools can exhaustively check them all. In understanding the performance of Txit, we have also uncovered several performance pathologies in TSX, knowledge of which will help other (potential) TSX users.

Conclusion

Reasoning about the correctness of concurrent and distributed systems has been a long-standing challenge. One of the fundamental reasons is the arbitrary space of interleavings of the systems, which rapidly grows in real-world systems, and becomes intractable for traditional verification and testing. Randomized concurrency testing (RCT) samples the interleavings to surface non-adversary and simple errors with significant probabilities. Without considering partial order semantics, however, RCT may sample partial-order equivalent interleavings redundantly. Such redundancies induce unwanted bias and weaken the probabilistic guarantees in testing real-world systems.

Towards practical and effective concurrency testing, we broke new ground to combine partial order semantics with RCT. We invented partial order sampling (POS), a new RCT framework with precise analysis of sampling event orderings, with a novel partial order method POS* to reduce the bias in sampling different partial orders. We proved its exponentially better probabilistic guarantees than state-of-the-art approaches. Our evaluation showed its effectiveness on generated and real-world multi-threaded benchmarks.

To apply and study POS in real-world concurrent systems, we built Morpheus to test implementations of distributed systems. Morpheus targets the high-level systems written in Erlang to focus on the protocol level errors, and introduces conflict analysis to further

eliminate redundant explorations regarding the partial order semantics. Our evaluation showed that Morpheus is effective in finding errors in popular real-world distributed systems in Erlang.

This thesis has presented novel and randomized partial order methods to explore concurrent programs with immense interleavings spaces and find non-adversary errors with strong guarantees. As real-world large systems involve not only concurrency, but handling failures and data input as well, reasoning about them requires further techniques and heuristics to deal with the combined complexity of the three aspects. In general, randomized methods can perform well in manifesting shallow conditions in a complicated search space, but such methods must be designed carefully to avoid biases degrading the worst-case guarantees for simple scenarios. Our work rigorously reduced such biases, resulted in strong guarantees and empirical performance in concurrency testing.

The implementations of partial order sampling and Morpheus are open-sourced online [75, 74].

Bibliography

[1] *ABA problem on Wikipedia*. http://en.wikipedia.org/wiki/ABA_problem.

[2] Parosh Abdulla et al. "Optimal Dynamic Partial Order Reduction." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 373–384. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535845. URL: http://doi.acm.org/10.1145/2535838.2535845.

[3] Elvira Albert et al. "Constrained Dynamic Partial Order Reduction." In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 392–410. ISBN: 978-3-319-96142-2.

[4] Elvira Albert et al. "Context-Sensitive Dynamic Partial Order Reduction." In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 526–543. ISBN: 978-3-319-63387-9.

[5] Ahmed Alquraan et al. "An Analysis of Network-partitioning Failures in Cloud Systems." In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 51–68. ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=3291168.3291173.

[6] *Amazon SimpleDB*. URL: https://aws.amazon.com/simpledb.

[7] *Apache Cassandra*. URL: http://cassandra.apache.org.

[8] Stavros Aronis, Scott Lystig Fritchie, and Konstantinos Sagonas. "Testing and Verifying Chain Repair Methods for Corfu Using Stateless Model Checking." In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 227–242. ISBN: 978-3-319-66845-1.

[9] Stavros Aronis et al. "Optimal Dynamic Partial Order Reduction with Observers." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. Cham: Springer International Publishing, 2018, pp. 229–248. ISBN: 978-3-319-89963-3.

[10] Radu Banabic and George Candea. "Fast Black-box Testing of System Recovery Code." In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 281–294. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168865. URL: http://doi.acm.org/10.1145/2168836.2168865.

[11] Christian Bienia. "Benchmarking Modern Multiprocessors." AAI3445564. PhD thesis. Princeton, NJ, USA, 2011. ISBN: 978-1-124-49186-8.

[12] Colin Blundell, E Christopher Lewis, and Milo MK Martin. "Subtleties of transactional memory atomicity semantics." In: *Computer Architecture Letters* 5.2 (2006), pp. 17–17.

[13] *Boost::Lockfree*. http://www.boost.org/doc/libs/1_55_0/doc/html/lockfree.html.

[14] Sebastian Burckhardt et al. "A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs." In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 167–178. ISBN: 978-1-60558-839-1. DOI: 10.1145/1736020.1736040. URL: http://doi.acm.org/10.1145/1736020.1736040.

[15] Richard Carlsson et al. "Core Erlang 1.0.3 language specification." In: (2004). URL: https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf.

[16] *CASSANDRA-6023*. URL: https://issues.apache.org/jira/browse/CASSANDRA-6023.

[17] M. Christakis, A. Gotovos, and K. Sagonas. "Systematic Testing for Detecting Concurrency Errors in Erlang Programs." In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Mar. 2013, pp. 154–163. DOI: 10.1109/ICST.2013.50.

[18] Koen Claessen et al. "Finding Race Conditions in Erlang with QuickCheck and PULSE." In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. Edinburgh, Scotland: ACM, 2009, pp. 149–160. ISBN: 978-1-60558-332-7. DOI: 10.1145/1596550.1596574. URL: http://doi.acm.org/10.1145/1596550.1596574.

[19] E. M. Clarke et al. "Symmetry reductions in model checking." In: *Computer Aided Verification*. Ed. by Alan J. Hu and Moshe Y. Vardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 147–158. ISBN: 978-3-540-69339-0.

[20] Lucas Cordeiro and Bernd Fischer. "Verifying Multi-threaded Software Using Smt-based Context-bounded Model Checking." In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 331–340. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985839. URL: http://doi.acm.org/10.1145/1985793.1985839.

[21] Pantazis Deligiannis et al. "Asynchronous Programming, Analysis and Testing with State Machines." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: ACM, 2015, pp. 154–164. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737996. URL: http://doi.acm.org/10.1145/2737924.2737996.

[22] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. "Delay-bounded Scheduling." In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 411–422. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926432. URL: http://doi.acm.org/10.1145/1926385.1926432.

[23] *Erlang Programming Language*. URL: https://erlang.org.

[24] Cormac Flanagan and Stephen N. Freund. "FastTrack: Efficient and Precise Dynamic Race Detection." In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 121–133. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542490. URL: http://doi.acm.org/10.1145/1542476.1542490.

[25] Cormac Flanagan and Patrice Godefroid. "Dynamic Partial-order Reduction for Model Checking Software." In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: ACM, 2005, pp. 110–121. ISBN: 1-58113-830-X. DOI: 10.1145/1040305.1040315. URL: http://doi.acm.org/10.1145/1040305.1040315.

[26] *Folly C++ library*. http://github.com/facebook/folly.

[27] Lars-Åke Fredlund and Hans Svensson. "McErlang: A Model Checker for a Distributed Functional Programming Language." In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP '07. Freiburg, Germany: ACM, 2007, pp. 125–136. ISBN: 978-1-59593-815-2. DOI: 10.1145/1291151.1291171. URL: http://doi.acm.org/10.1145/1291151.1291171.

[28] Patrice Godefroid. "Model Checking for Programming Languages Using VeriSoft." In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: ACM, 1997, pp. 174–186. ISBN: 0-

89791-853-3. DOI: 10.1145/263699.263717. URL: http://doi.acm.org/10.1145/263699.263717.

[29] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Ed. by J. van Leeuwen, J. Hartmanis, and G. Goos. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996. ISBN: 3540607617.

[30] Huayang Guo et al. "Practical Software Model Checking via Dynamic Interface Reduction." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 265–278. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043582. URL: http://doi.acm.org/10.1145/2043556.2043582.

[31] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Mar. 2014.

[32] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. June 2014.

[33] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006. ISBN: 0262101149.

[34] Charles Edwin Killian et al. "Mace: Language Support for Building Distributed Systems." In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: ACM, 2007, pp. 179–188. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250755. URL: http://doi.acm.org/10.1145/1250734.1250755.

[35] Kyle Kingsbury. *Jepsen*. 2016-2019. URL: http://jepsen.io/.

[36] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[37] Tanakorn Leesatapornwongsa et al. "SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems." In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 399–414. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685080.

[38] Tanakorn Leesatapornwongsa et al. "TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems." In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 517–530. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872374. URL: http://doi.acm.org/10.1145/2872362.2872374.

[39] *liblfds*. http://liblfds.org.

[40] Haopeng Liu et al. "DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems." In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 677–691. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037735. URL: http://doi.acm.org/10.1145/3037697.3037735.

[41] Haopeng Liu et al. "FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems." In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA: ACM, 2018, pp. 419–431. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3177161. URL: http://doi.acm.org/10.1145/3173162.3177161.

[42] Shan Lu et al. "AVIO: Detecting Atomicity Violations via Access Interleaving Invariants." In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: ACM, 2006, pp. 37–48. ISBN: 1-59593-451-0. DOI: 10.1145/1168857.1168864. URL: http://doi.acm.org/10.1145/1168857.1168864.

[43] Jeffrey F. Lukman et al. "FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems." In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: ACM, 2019, 20:1–20:16. ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303986. URL: http://doi.acm.org/10.1145/3302424.3303986.

[44] Rupak Majumdar and Filip Niksic. "Why is Random Testing Effective for Partition Tolerance Bugs?" In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 46:1–46:24. ISSN: 2475-1421. DOI: 10.1145/3158134. URL: http://doi.acm.org/10.1145/3158134.

[45] Dahlia Malkhi et al. "From Paxos to CORFU: A Flash-speed Shared Log." In: *SIGOPS Oper. Syst. Rev.* 46.1 (Feb. 2012), pp. 47–51. ISSN: 0163-5980. DOI: 10.1145/2146382.2146391. URL: http://doi.acm.org/10.1145/2146382.2146391.

[46] Maged M Michael. "ABA prevention using single-word instructions." In: *IBM Research Division, RC23089 (W0401-136), Tech. Rep* (2004).

[47] Madanlal Musuvathi and Shaz Qadeer. "Iterative Context Bounding for Systematic Testing of Multithreaded Programs." In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: ACM, 2007, pp. 446–455. ISBN: 978-1-59593-633-2. DOI: 10.1145/

1250734.1250785. URL: http://doi.acm.org/10.1145/1250734.1250785.

[48] Madanlal Musuvathi et al. "Finding and Reproducing Heisenbugs in Concurrent Programs." In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 267–280. URL: http://dl.acm.org/citation.cfm?id=1855741.1855760.

[49] Mayur Naik, Alex Aiken, and John Whaley. "Effective Static Race Detection for Java." In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. Ottawa, Ontario, Canada: ACM, 2006, pp. 308–319. ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1134018. URL: http://doi.acm.org/10.1145/1133981.1134018.

[50] *nbds*. http://nbds.googlecode.com.

[51] Huyen T. T. Nguyen et al. "Quasi-Optimal Partial Order Reduction." In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 354–371. ISBN: 978-3-319-96142-2.

[52] Robert O'Callahan and Jong-Deok Choi. "Hybrid Dynamic Data Race Detection." In: *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '03. San Diego, California, USA: ACM, 2003, pp. 167–178. ISBN: 1-58113-588-2. DOI: 10.1145/781498.781528. URL: http://doi.acm.org/10.1145/781498.781528.

[53] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm." In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 305–320. ISBN: 978-1-931971-10-2. URL: http://dl.acm.org/citation.cfm?id=2643634.2643666.

[54] Burcu Kulahcioglu Ozkan et al. "Randomized Testing of Distributed Systems with Probabilistic Guarantees." In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 160:1–160:28. ISSN: 2475-1421. DOI: 10.1145/3276530. URL: http://doi.acm.org/10.1145/3276530.

[55] Soyeon Park, Shan Lu, and Yuanyuan Zhou. "CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places." In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: ACM, 2009, pp. 25–36. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508249. URL: http://doi.acm.org/10.1145/1508244.1508249.

[56] *Pyevolve*. http://pyevolve.sourceforge.net/.

[57] Robbert van Renesse and Fred B. Schneider. "Chain Replication for Supporting High Throughput and Availability." In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 7–7. URL: http://dl.acm.org/citation.cfm?id=1251254.1251261.

[58] Koushik Sen. "Effective Random Testing of Concurrent Programs." In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: ACM, 2007, pp. 323–332. ISBN: 978-1-59593-882-4. DOI: 10.1145/1321631.1321679. URL: http://doi.acm.org/10.1145/1321631.1321679.

[59] Koushik Sen. "Race Directed Random Testing of Concurrent Programs." In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 11–21. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375584. URL: http://doi.acm.org/10.1145/1375581.1375584.

[60] Jiri Simsa, Randy Bryant, and Garth Gibson. "dBug: Systematic Evaluation of Distributed Systems." In: *Proceedings of the 5th International Conference on Systems Software Verification*. SSV'10. Voncouver, BC, Canada: USENIX Association, 2010, pp. 3–3. URL: http://dl.acm.org/citation.cfm?id=1929004.1929007.

[61] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. "SMC: A Symmetry-based Model Checker for Verification of Safety and Liveness Properties." In: *ACM Trans. Softw. Eng. Methodol.* 9.2 (Apr. 2000), pp. 133–166. ISSN: 1049-331X. DOI: 10.1145/350887.350891. URL: http://doi.acm.org/10.1145/350887.350891.

[62] *The Go Programming Language Specification*. URL: https://golang.org/ref/spec.

[63] Paul Thomson, Alastair F. Donaldson, and Adam Betts. "Concurrency Testing Using Controlled Schedulers: An Empirical Study." In: *ACM Trans. Parallel Comput.* 2.4 (Feb. 2016), 23:1–23:37. ISSN: 2329-4949. DOI: 10.1145/2858651. URL: http://doi.acm.org/10.1145/2858651.

[64] *Transactional Synchronization in Haswell*. https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell. 2012.

[65] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. "RELAY: Static Race Detection on Millions of Lines of Code." In: *Proceedings of the the 6th Joint Meeting of the Eu-*

ropean Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ESEC-FSE '07. Dubrovnik, Croatia: ACM, 2007, pp. 205–214. ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287654. URL: http://doi.acm.org/10.1145/1287624.1287654.

[66] Chao Wang, Mahmoud Said, and Aarti Gupta. "Coverage Guided Systematic Concurrency Testing." In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 221–230. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985824. URL: http://doi.acm.org/10.1145/1985793.1985824.

[67] *Why WhatsApp Only Needs 50 Engineers for Its 900M Users*. URL: https://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/.

[68] Ulf Wiger. *GitHub - locks*. 2017. URL: https://github.com/uwiger/locks.git.

[69] *Wikipedia: Read-copy-update*. https://en.wikipedia.org/wiki/Read-copy-update.

[70] Steven Cameron Woo et al. "The SPLASH-2 Programs: Characterization and Methodological Considerations." In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ISCA '95. S. Margherita Ligure, Italy: ACM, 1995, pp. 24–36. ISBN: 0-89791-698-0. DOI: 10.1145/223982.223990. URL: http://doi.acm.org/10.1145/223982.223990.

[71] Junfeng Yang et al. "MODIST: Transparent Model Checking of Unmodified Distributed Systems." In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, pp. 213–228. URL: http://dl.acm.org/citation.cfm?id=1558977.1558992.

[72] Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. *Inspect: A runtime model checker for multithreaded C programs*. Tech. rep. Salt Lake City, UT, USA, 2008.

[73] Jie Yu et al. "Maple: A Coverage-driven Testing Tool for Multithreaded Programs." In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. Tucson, Arizona, USA: ACM, 2012, pp. 485–502. ISBN: 978-1-4503-1561-6. DOI: 10.1145/2384616.2384651. URL: http://doi.acm.org/10.1145/2384616.2384651.

[74] Xinhao Yuan. *Morpheus*. 2019. URL: https://github.com/xinhaoyuan/morpheus.

[75] Xinhao Yuan. *Partial order sampling: benchmark code*. 2018. URL: https://github.com/xinhaoyuan/pos-benchmarks.

[76] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. "Partial Order Aware Concurrency Sampling." In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 317–335. ISBN: 978-3-319-96142-2.