Trace Aware Random Testing for Distributed Systems

BURCU KULAHCIOGLU OZKAN, Max Planck Institute for Software Systems (MPI-SWS), Germany
RUPAK MAJUMDAR, Max Planck Institute for Software Systems (MPI-SWS), Germany
SIMIN ORAEE, Max Planck Institute for Software Systems (MPI-SWS), Germany

Distributed and concurrent applications often have subtle bugs that only get exposed under specific schedules. While these schedules may be found by systematic model checking techniques, in practice, model checkers do not scale to large systems. On the other hand, naive random exploration techniques often require a very large number of runs to find the specific interactions needed to expose a bug. In recent years, several random testing algorithms have been proposed that, on the one hand, exploit state-space reduction strategies from model checking and, on the other, provide *guarantees* on the probability of hitting bugs of certain kinds.

These existing techniques exploit two orthogonal strategies to reduce the state space: *partial-order reduction* and *bug depth*. Testing algorithms based on partial order techniques, such as RAPOS or POS, ensure non-redundant exploration of independent interleavings among system events by imposing an equivalence relation on schedules and ideally exploring only one schedule from each equivalence class. Techniques based on bug depth, such as PCT, exploit the empirical observation that many bugs are exposed by the clever scheduling of a small number of key events. They bias the sample space of schedules to only cover all executions of small depth, rather than the much larger space of all schedules. At this point, there is no random testing algorithm that combines the power of both approaches.

In this paper, we provide such an algorithm. Our algorithm, *trace-aware* PCT (taPCT), extends and unifies several different algorithms in the random testing literature. It samples the space of low-depth executions by constructing a schedule *online*, while taking dependencies among events into account. Moreover, the algorithm comes with a theoretical guarantee on the probability of sampling a trace of low depth—the probability grows exponentially with the depth but only polynomially with the number of racy events explored. We further show that the guarantee is optimal among a large class of techniques.

We empirically compare our algorithm with several state-of-the-art random testing approaches for concurrent software on two large-scale distributed systems, Zookeeper and Cassandra, and show that our approach is effective in uncovering subtle bugs and usually outperforms related random testing algorithms.

CCS Concepts: • **Mathematics of computing → Combinatorics**; • **Software and its engineering → Software testing and debugging**; • **Theory of computation** → *Generating random combinatorial structures*.

Additional Key Words and Phrases: distributed systems, random testing, hitting families, partial order reduction

ACM Reference Format:
Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. 2019. Trace Aware Random Testing for Distributed Systems. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 180 (October 2019), 29 pages. https://doi.org/10.1145/3360606

Authors' addresses: Burcu Kulahcioglu Ozkan, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Str. 26, Kaiserslautern, Rheinland-Pfalz, 67663, Germany, burcu@mpi-sws.org; Rupak Majumdar, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Str. 26, Kaiserslautern, Rheinland-Pfalz, 67663, Germany, rupak@mpi-sws.org; Simin Oraee, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Str. 26, Kaiserslautern, Rheinland-Pfalz, 67663, Germany, siminoraee@mpi-sws.org.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART180
https://doi.org/10.1145/3360606

1 INTRODUCTION

We consider the problem of systematically testing distributed message-passing programs. Such programs are difficult to get right because the programmer has to reason about a large number of dependencies between messages and bugs may occur under very specific interleavings of the executing processes. Systematic testing aims at exploring the space of interleavings by controlling the underlying scheduler. The space of interleavings grows exponentially with the length of execution; therefore, a key challenge in systematic testing is to control the set of possible interleavings explored, while still guaranteeing that many bugs will be found.

Formal techniques, such as model checking, provide different heuristics to reduce the state space and are complete in the limit. However, they seldom finish exploring all schedules of a large concurrent system within reasonable testing budgets. Moreover, since they are usually based on deterministic search strategies, a time-bounded exploration may end up exploring only a local part of the state space. In recent years, a number of testing algorithms have explored systematic randomization as an effective way to sample from large state spaces [Burckhardt et al. 2010; Kulahcioglu Ozkan et al. 2018; Majumdar and Niksic 2018; Nagarakatte et al. 2012; Sen 2007; Yuan et al. 2018]. A key feature of these algorithms is that they come with *theoretical guarantees* of success: for each schedule of interest, they provide a minimal probability with which that schedule will be picked by the search. The design of the algorithms, and their theoretical analysis, relies on structural properties of the state space to reduce the space of schedules. Two particularly effective approaches in this vein have been *trace-aware random testing* and *depth-bounded random testing*.

Trace-aware random testing: RAPOS *and* POS. Trace-aware random testing algorithms are based on partial order reduction techniques [Abdulla et al. 2014; Flanagan and Godefroid 2005; Godefroid 1996, 1997]. They exploit the fact that many interleavings in a distributed system are equivalent to each other, because they only differ in the ordering of independent events. Thus, these algorithms attempt to sample schedules from different equivalence classes, or *traces*.

Sen [Sen 2007] introduced *randomized partial order sampling* (RAPOS), a randomized exploration technique with the aim to cover traces "more uniformly" than pure random walk. Instead of a single enabled event, RAPOS picks a random subset of pairwise independent events and schedules all of them in an arbitrary order. Since mutual ordering between these events are irrelevant in exploring traces, RAPOS can perform better than naive random search. RAPOS did not come with a formal guarantee on the uniformity of the search. Recently, Yuan et al. [Yuan et al. 2018] improve upon RAPOS by presenting a new algorithm, POS, for trace-aware random sampling that comes with non-trivial probability bounds. Yuan et al. demonstrate the effectiveness of POS (over RAPOS) on a set of small benchmarks. They also provide a lower bound on the probability that a particular trace will be picked by the random scheduler. Unfortunately, the lower bound is (essentially) exponentially small in the length of the execution; as we discuss later, this is unavoidable for *any* online testing algorithm.

Depth-bounded random testing: PCT *and* PCTCP. Depth-bounded random testing algorithms exploit the empirical observation that many bugs in concurrent systems are exposed by considering the sequencing of a small number of events—the "bug depth"—independent of the ordering of the rest of the events [Burckhardt et al. 2010; Leesatapornwongsa et al. 2016]. Depth-bounded testing algorithms, such as PCT for multithreaded shared memory programs [Burckhardt et al. 2010] or PCTCP for distributed message-passing programs [Kulahcioglu Ozkan et al. 2018], take the depth as a parameter and sample a subset of schedules which is sufficient to cover all executions which fall within the bug depth. Their probabilistic guarantees reduce exponentially with the (small)

depth bound rather than with the (large) length of executions. These algorithms have been shown to be effective in finding bugs in multithreaded and distributed programs.

Unfortunately, these algorithms do not take into account the commutativity of independent events. Thus, when there are many independent events in a system, these algorithms can perform poorly. On the other hand, POS and RAPOS do not prioritize small depth, and can often "get stuck." In summary, state-of-the-art random testing algorithms either exploit traces (POS) or exploit the bug depth (PCT). It was not known how to combine these two features at the same time, while maintaining theoretical guarantees.

taPCT*: A depth-bounded, trace-aware algorithm.* In this paper, we develop taPCT, a *trace-aware* and *depth-bounded* random testing algorithm for distributed message-passing programs. Our algorithm generalizes and unifies PCT and POS: it uses the insights of PCT to bias the sampling to prioritize schedules of small depth while at the same time exploiting the independence relation on events. In particular, we show how PCT and POS can be obtained as a special cases of our algorithm.

Like PCT and unlike POS, taPCT samples from a subset of schedules, called a *strong d-hitting family*, that is sufficient to cover all bugs of depth d. At the same time, like POS and unlike PCT, it takes the dependence structure of the events into account, deciding to change priorities only on potentially racy events.

Overall, the taPCT algorithm is simple: it involves maintaining prioritized chains of events online (as in PCT), where the priorities are assigned randomly, picking highest priority events for execution at all times, and reducing the priorities of some chains at $d-1$ randomly chosen *racy* events in the execution (as in POS). The probabilistic guarantees of taPCT follow from the corresponding guarantees for PCT: the probability to hit a trace is exponential in the depth bound but only polynomial in the number of racy events in the execution.

The taPCT algorithm subsumes previous random testing algorithms. When the number of racy events increases to the number of events, we obtain PCT. In the special case, where each event is its own chain, we obtain a novel depth-bounded version of POS, called d−POS. Finally, when each event is in its own chain and the depth increases to the number of events, we obtain POS.

Evaluation. We have implemented taPCT for distributed message passing applications and evaluated it on two large-scale distributed systems: Zookeeper and Cassandra. The concurrency bugs in such distributed systems are known to be "deep," i.e., triggering the bug requires some certain ordering of a large number of messages [Leesatapornwongsa et al. 2016]. Therefore, it is difficult to detect these bugs by naive random testing methods.

We compare taPCT against POS, PCT, d−POS, and RAPOS on two metrics: uniformity of sampling and efficacy in bug finding. Our evaluation shows that these algorithms do not differ substantially on uniformity of sampling (on our benchmarks). However, taPCT outperforms all other algorithms in finding bugs, e.g., by reproducing known bugs.

Our conclusion is that taPCT brings together the advantages of both trace aware sampling and depth bounded sampling.

2 OVERVIEW

2.1 Model of Distributed Systems

We consider a simple model of distributed systems composed of a fixed number of distributed *nodes* that run concurrently with each other. The nodes maintain their own local states and communicate with each other only by exchanging *messages*. At each point, each node maintains a bag of enabled messages to be executed. A scheduler picks a node and an enabled message and executes it atomically.

Executing a message produces new messages in the system, either to the node itself or to a different node. In addition to protocol messages in the system, messages also model events such as node crash or node restart.

Let $\texttt{Nodes}$ be the set of nodes and $\texttt{Msgs}$ be the set of all messages exchanged in the system. We define *events* as tuples $\langle \texttt{recv}, \texttt{send}, \texttt{msg} \rangle$ for $\texttt{recv}, \texttt{send} \in \texttt{Nodes}$ and $\texttt{msg} \in \texttt{Msgs}$; for an event e, $\texttt{recv}(e)$ is the receiver node of the message, $\texttt{send}(e)$ the sender node (possibly the same as the receiver), and $\texttt{msg}(e)$ is the message. Let Σ be the set of events.

A *state* of the system is a map $s : \texttt{Nodes} \mapsto 2^{\Sigma}$ from nodes to sets of enabled events. A transition in the system corresponds to picking a node $\texttt{node}$ and an event $e \equiv \langle \cdot, \cdot, \texttt{msg} \rangle \in s(\texttt{node})$ and executing the message. Executing the message $e = \langle \texttt{node}, \cdot, \cdot \rangle$ by can lead to the creation of new events $e_i \equiv \langle \texttt{node}_i, \texttt{node}, \texttt{msg}_i \rangle$, i.e., $\texttt{node}$ may send new messages to some nodes upon processing e. We say that each e_i *causally depends* on e. The new state s' is obtained by removing e from $s(\texttt{node})$ and adding e_i to $s(\texttt{node}_i)$ for each i, and we write $s \xrightarrow{\texttt{node}:e} s'$.

An *execution* is a sequence

$$s_0 \xrightarrow{\texttt{node}_0:e_0} s_1 \xrightarrow{\texttt{node}_1:e_1} \dots \xrightarrow{\texttt{node}_n:e_n} s_{n+1}$$

of states s_i and events e_i executed by $\texttt{node}_i$. We call the sequence $\langle \texttt{node}_0 : e_0 \rangle \dots \langle \texttt{node}_n : e_n \rangle$ a *schedule*.

A schedule induces a partial ordering among events; the ordering is captured by a binary *dependence* relation $D \subseteq \Sigma \times \Sigma$.

Definition 2.1 (Dependence Relation). Let e_i and e_j be respectively the ith and jth events in a schedule. The two events e_i and e_j are dependent, $(e_i, e_j) \in D$ iff:

either (i) $\exists k : i \leq k < j$ such that $\texttt{recv}(e_i) = \texttt{recv}(e_k)$ and e_j is transitively causally dependent on e_k;

or (ii) $\texttt{recv}(e_i) = \texttt{recv}(e_j)$.

Informally, the case 2.1(i) captures (transitive) causal dependency between events, where one event potentially causes (enables) another event. Causally dependent events are not both enabled at the same time and they cannot be reordered. The second case 2.1(ii) captures events that may be co-enabled and executed at the same node; different processing order of these events may result in different outcomes. In an execution, we call an event a *racy* event if there exists another event that is both concurrently enabled with it and also dependent to it. Two events which are not dependent are *independent*. The execution of two independent events do not affect each other.

From a shedule $w = (\texttt{node}_1 : e_1) \dots (\texttt{node}_n : e_n)$ we define a labeled (Mazurkiewicz) *trace* graph $\langle w \rangle_D = (V, E, \ell)$ as:

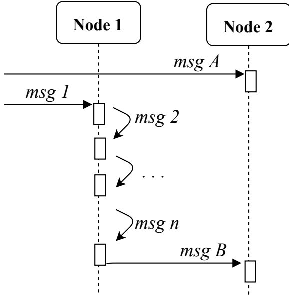
- $V = \{1, \dots, n\}$, the set of positions of the schedule, and the label ℓ maps i to $(\texttt{node}_i : e_i)$;
- $(i, j) \in E$ iff $i < j$ and e_j is dependent to e_i.

The reflexive transitive closure of the edge relation is acyclic, and captures the ordering constraints on a schedule. We shall abstract away from the identities of nodes and only consider isomorphism classes of trace graphs.

Trace graphs induce an equivalence on schedules: two schedules are equivalent if they have the same trace graph. Intuitively, we can swap the position of any two consecutive independent messages. We call a testing algorithm is *trace-aware* if it samples the family of trace graphs, rather than the family of underlying schedules.

Example 2.2. Figure 1(a) shows the sequence of messages exchanged in a distributed system that has two nodes Node 1 and Node 2. Initially, two concurrent messages *msg* 1 and *msg* A are waiting

(a) Sequence diagram of the execution



(b) Partial order

$$msg\ 1 \longrightarrow \ldots \longrightarrow msg\ n \longrightarrow msg\ B$$

$$msg\ A$$

(c) Probability of scheduling *msg A* after *msg B*:

Random walk		$1/2^{n+1}$
POS	(trace aware)	$1/(n+2)$
PCT	(d-bounded)	$1/2$
d−POS	(trace aware + d-bounded)	$1/(n+2)$
taPCT	(trace aware + d-bounded)	$1/2$

Fig. 1. A sample execution where the event of processing *msg A* is dependent to *msg B*. A concurrency bug occurs when *msg A* is executed by Node 2 after *msg B*.

at Node 1 and Node 2, respectively. When Node 1 processes *msg* 1, it sends a message *msg* 2, to itself. After a chain of n such self messages, it sends *msg B* to Node 2.

Figure 1(b) shows the resulting partial order. The initial events *msg* 1 and *msg A* are not causally dependent to each other, and hence they are independent. The messages *msg* 1, . . ., *msg n*, and *msg B* are dependent, since the new messages are causally dependent to the processed message. The messages *msg A* and *msg B* are concurrently enabled with the same receiver, hence, they are racy. Therefore, the order of processing these events by Node 2 may result in different states. In the following, we shall assume that the program has a concurrency bug that is found only when *msg B* is processed before *msg A*. □

2.2 Bug Depth

Burckhardt et al. [Burckhardt et al. 2010] introduced the notion of *bug depth* to characterize concurrency bugs. The bug depth measures, informally, the minimal number of ordering constraints among events that a schedule must enforce in order to find a buggy execution. For example, the buggy execution in Example 2.2 has bug depth one: the crucial ordering constraint is that *msg B* comes before *msg A*.

The notion of bug depth is made formal using *strongly hitting* schedules for a tuple of events Burckhardt et al. [2010]; Kulahcioglu Ozkan et al. [2018]. Roughly, a schedule *strongly hits* a d-tuple of events $(e_0 \ldots, e_{d-1})$ if these d events are scheduled in the order they appear in the tuple and as late as possible in the execution. A strong d-hitting family of schedules is a subset of schedules that has the property that every execution of depth d (identified by a d-tuple of events) is strongly hit by some schedule in the family.

2.3 Randomized Testing Algorithms: State of the Art

The ideal goal of testing is to explore every trace graph produced by a distributed system. Since the number of graphs is too enormous for exhaustive exploration, testing approaches usually *randomly sample* a suitable *subset* of the space of trace graphs.

Random walk. The simplest randomized sampling algorithm picks an enabled event uniformly at random at each step. Unfortunately, since it ignores both the dependence structure and bug depth, simple random sampling may perform poorly: for Example 2.2, the probability of hitting the bug is

$1/2^{n+1}$. This is because the algorithm has to choose between *msg A* and *msg i* for $i = 1, \ldots, n$ and *msg B*, and the probability that it picks A only after the other $n + 1$ messages is $1/2^{n+1}$.

We now revisit two state-of-the-art randomized algorithms with probabilistic guarantees: POS [Yuan et al. 2018] and PCT [Kulahcioglu Ozkan et al. 2018]. These algorithms improve random sampling in orthogonal directions: POS takes into account dependencies but not bug depth; PCT takes into account bug depth but not dependencies. Our goal later will be to combine the good features of these algorithms. We compare the "quality" of the algorithms in terms of the *minimal probabilistic guarantee* of picking a trace from a given sub-family Scheds of schedules (e.g., all schedules or d-hitting schedules):

$$\min_{w \in \mathsf{Scheds}} \Pr[\text{ tester executes } \langle w \rangle_D] \tag{1}$$

(We discuss in Section 3 why this notion is appropriate.)

POS*: Trace-aware sampling [Yuan et al. 2018].* The POS algorithm is based on a priority based scheduler which uses the dependence information between events. It assigns an initial random priority to each event; this determines a random permutation of all events. At each step, POS picks and executes the enabled event with the highest priority. Then, it updates the priorities of all events that are dependent with the scheduled event, while preserving the priorities of the other events. This creates a new random permutation of all events.

The POS algorithm samples the buggy schedule in Example 2.2 with probability $1/(n + 2)$. It assigns random priorities to each single event and schedules an event with the highest priority. In that example, none of the event priorities are updated until *msg A* or *msg B* are scheduled, since there exist no dependent enabled events to the scheduled event. The probability to schedule *msg A* after *msg B*, in other words the probability of *msg A* to have the smallest probability over all $n + 2$ events is $1/(n + 2)$. This is exponentially better than random walk.

Theoretically, POS provides a guarantee of picking every trace graph with a minimal probability that falls exponentially with the number of events. Thus, in the worst case, the guarantees are similar to a random walk, but in practice, POS often performs much better. As we will explain in Section 3.2, a randomized algorithm that samples from the set of all possible executions of the program, cannot guarantee any better probability than an exponentially small probability in the number of events.

PCT*: Depth-bounded sampling [Burckhardt et al. 2010; Kulahcioglu Ozkan et al. 2018].* [1] Depth-bounded sampling algorithms take an additional depth bound parameter d as input and pick schedules randomly from a *strong d-hitting family*. Briefly, these algorithms randomly choose a d-tuple of events online in the execution and build a schedule so that it strongly hits the d-tuple of events.

The sampling procedure in PCT uses a priority scheme similar to POS, but ignores independence between events. The key innovation in PCT is the maintenance of trace graphs using their *chain partitioning*, and random sampling based on chains rather than individual events. A chain partitioning of a partial order represents the order as a disjoint union of *chains* (linear orderings of elements). For example, in Figure 1(b), the partial order can be represented by two chains, part 1 and part 2 in the figure.

The PCT algorithm maintains chain partitions through an online chain partitioning algorithm and assigns random priorities to chains. In each step, it schedules the next enabled event in the

[1] The original PCT algorithm [Burckhardt et al. 2010] was described for multithreaded shared memory programs. It was generalized to message-passing systems and called PCTCP [Kulahcioglu Ozkan et al. 2018]. In the following, we write "PCT" for the generalized algorithm, since the two algorithms are similar in spirit and differ only in technical aspects.

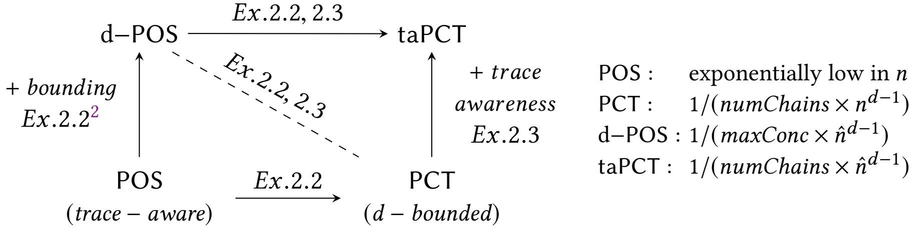


Fig. 2. The relationship between the randomized algorithms in terms of their probability guarantees for sampling a certain schedule. The parameter *numChains* is the number of chains of events, *maxConc* is the maximum number of events concurrently enabled with an event, $\hat{n}$ is the number of racy events, n is the total number of events, and d is the bug depth. The arrows in the figure point from an algorithm with a lower probability guarantee to an algorithm with a higher one, which are demonstrated by the examples in the labels. The algorithms connected by the dashed line are incomparable w.r.t. their guarantees.

highest priority chain. Additionally, it selects $d - 1$ random *priority change points* at which the priority of a chain is decreased. A combinatorial argument [Kulahcioglu Ozkan et al. 2018] shows that this procedure selects each depth-d schedule with sufficiently high probability.

For the depth-1 bug in Example 2.2, PCT strongly hits the schedule with probability $\frac{1}{2}$. This is because a chain partitioning of the partial order (Figure 1(b)) has two chains and the bug is hit depending on the initial chain priorities of Chains 1 and 2. Since $d = 1$, there are no priority change points in this example.

As we will discuss in more detail in Section 4, PCT samples a schedule of depth d (which strongly hits certain d events), with a probability of at least $1/(numChains \times n^{d-1})$ where *numChains* is the number of chains in the chain partitioning and n is the total number of events. For small d, this is exponentially better than random walk; however, PCT does not provide any guarantee for bugs of depth greater than d. Since PCT does not use the dependency relation between events, it is possible that a strongly hit event is independent of all other events concurrently enabled with it. This can result in redundant sampling of traces.

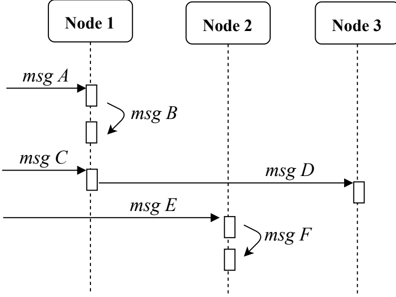
2.4 Our Contribution: taPCT = Depth bounded + Trace aware

In this section, we introduce *trace-aware* PCT (taPCT), a randomized sampling algorithm. It brings together trace awareness and bounded sampling. Overtly, taPCT is a modification and generalization of PCT to consider dependencies among events. It turns out that this generalization also subsumes the POS algorithm, with or without a depth bound.

The taPCT algorithm modifies PCT by considering the dependency relation while selecting the events to be strongly hit. Similar to PCT, taPCT strongly hits a tuple of events $(e_0 \ldots, e_{d-1})$ based on a chain partitioning of events. Different from PCT, taPCT selects the $d - 1$ events to strongly hit from the set of *racy* events: those events that have some concurrently enabled and dependent events. Simply, it strongly hits an event e only when strongly hitting e may result in a different trace. The formal guarantee of taPCT improves that of PCT as it selects the $d - 1$ events from the smaller set of $\hat{n}$ *racy* events, instead of the total number n of events. (Trivially, $\hat{n} \leq n$ and in practice,

[2]We consider a variant of Example 2.2 in Figure 1 to demonstrate the performance of d−POS over POS. Different from Example 2.2, assume that not only *msg B* is dependent to *msg A* but all the events *msg* 1 to *msg n* are also dependent to *msg A*. In that case, POS updates the priorities of the events after the execution of each event and hits the bug with the probability of $\frac{1}{2^{n+1}}$, exponentially low in the number of events. However, d−POS with $d = 1$ hits the bug by sampling a schedule which strongly hits *msg A* with a probability of $\frac{1}{n+2}$.

(a) Sequence diagram of the execution



(b) Chain partitioning

Chain 1: $msgA \longrightarrow msgB$
Chain 2: $msgC \longrightarrow msgD$
Chain 3: $msgE \longrightarrow msgF$

(c) Probability of strongly hitting d-tuple ($msg\ C$, $msg\ B$):

PCT : $1/(numChains \times n^{d-1}) = 1/(3 \times 6) = 1/18$
d−POS : $1/(maxConc \times \hat{n}^{d-1}) = 1/(5 \times 3) = 1/15$
taPCT : $1/(numChains \times \hat{n}^{d-1}) = 1/(3 \times 3) = 1/9$

Fig. 3. A sample distributed system execution where the event of processing *msg A* and *msg B* are dependent to and concurrently enabled with *msg C*. A concurrency bug occurs when *msg C* is executed between *msg A* and *msg B*. 3b shows the causally dependent chains of events and 3c lists the probability guarantees provided by the depth bounded algorithms.

the number of racy events can be many fewer than the number of all events.) taPCT samples a schedule of depth d, with a probability of at least $numChains \times (\hat{n})^{d-1}$.

It turns out that we get another algorithm, d−POS, which can be seen simultaneously as a modification of POS to sample from strong d-hitting families of schedules as well as the special case of taPCT when we use a trivial chain partitioning that assigns a new chain to each event. d−POS assigns random priorities to every single event. Different from POS, it updates the priorities of only a randomly selected subset of $d - 1$ racy events $(e_1 \ldots, e_{d-1})$ so that it schedules these events in a certain order. The priority value of an event e_i in the $(d-1)$-tuple is updated so that the order of the events in the $(d-1)$-tuple are preserved in the produced schedule. We will discuss the algorithm more detail in Section 4 and show that d−POS samples a schedule of depth d (which strongly hits certain d events) with a probability of at least $maxConc \times n^{d-1}$ where n is the total number of events and $maxConc$ is the maximum number of co-enabled events.

The relationship between the randomized algorithms is shown in Figure 2. The direction of the arrow shows the superiority between a pair of algorithms in terms of guaranteed probability of sampling a certain strongly d-hitting schedule. The d−POS and PCT algorithms provide a better guarantee than POS, which is exponentially low only in the constant depth bound d. The guarantees of d−POS and PCT are pairwise incomparable, as it depends on the relation of events in an execution. The taPCT algorithm provides a better guarantee than both d−POS and PCT, owing to better chain partitioning and better trace awareness respectively.

Example 2.3. Figure 3a shows an example execution of a distributed system with three nodes. Node 1 receives two concurrent messages *msg A* and *msg C*. Upon processing *msg A*, it sends *msg B* to itself, and upon processing *msg C* it sends *msg D* to Node 3. Node 2 receives *msg E* and sends *msg F* to itself upon processing it. As shown in 3b, the execution can be partitioned into three chains. The initial messages *msg A*, *msg B*, and *msg C* are all concurrent to each other and placed in different chains. The messages created by processing a message are appended to the chains of the messages they are causally dependent to. In this example, the events *msg A* and *msg C* are racy to each other, i.e., their different orders may yield a different state in Node 1. Similarly, *msg B* and *msg C* are racy, and there are no other racy events.

Let us consider a concurrency bug that is exposed when the events *msg A*, *msg C*, and *msg B* are executed in this order. Then, a schedule which strongly hits the tuple (*msg C*, *msg B*) exposes the bug. Given an initial assignment of event or chain priorities, strongly hitting the pair requires changing the priority of a single event. More specifically, an execution which assigns *msg C* the smallest priority among its racy events and changes the priority of the event *msg B* to execute it after *msg C* exposes the bug. In the following paragraphs, we compare the depth bounded algorithms d−POS, PCT, and taPCT in terms of their probability of strongly hitting (*msg C*, *msg B*).

For Example 2.2, d−POS provides a lower guarantee than PCT. There are no priority changes and the algorithms' performance depends only on the probability of assigning the smallest priority to *msg A*, among *numChains* or *maxConc* number of priorities. The number of chains (*numChains*) is 2, since most of the events are causally dependent to each other. However, the maximum number of concurrently enabled events is $n + 2$ since *msg A* is concurrently enabled with all $n + 1$ events in Node 1.

On the other hand, for the program in Example 2.3 (Figure 3a), d−POS provides a higher guarantee than PCT. The buggy schedule in this example requires assigning the smallest priority to *msg C* as well as changing the priority of *msg B* after executing *msg A*. d−POS selects *msg B* as the priority change point with a probability of 1/3. This is because it selects the priority change point from the set of racy events, which have some concurrently enabled dependent events to them. In this example, $\hat{n} = 3$ events are racy (*msg A*, *msg B* and *msg C*). On the other hand, PCT selects *msg B* as the priority change point with a probability of 1/6, selected uniformly at random from the set of all events.

The taPCT algorithm improves d−POS in the process of strongly hitting the first event e_0, i.e., selecting the event with smallest initial priority. Similar to d−POS, it selects this event among the racy events. Moreover, it exploits the chain partitioning in PCT and provides a larger probability for assigning the smallest priority to *msg C*. Simply, it selects the smallest priority among the *chains* instead of *events*. Hence, a schedule sampled by taPCT strongly hits all the events in the smallest priority chain, while a schedule sampled by d−POS strongly hits a single smallest priority event.

In summary, both taPCT and its special case d−POS combine trace-awareness and depth-bounding. There is a small theoretical edge to taPCT in terms of the minimal probability of picking a trace. As we show in Section 5, taPCT outperforms the other algorithms empirically as well, in terms of bug finding.

2.5 Implementation Challenges

The taPCT algorithm assumes the set of events can be controlled by the scheduler. This requires an instrumentation of the system under test, which can be nontrivial for a complicated system. There are two further challenges.

First, we assume that the set of racy events is known. It is difficult to classify an event to be racy while the algorithm is executing, because the witnessing event that is dependent to it may not exist at the point the taPCT algorithm processes the event. Our solution is to perform a preliminary dynamic analysis to approximate a set of racy events.

Second, the dependency relation that identifies all events at a node to be dependent can be overly coarse. The performance of a trace-aware algorithm is dependent on the precision of the dependency relation between the events as the algorithm exploits the commutativity of independent events to eliminate redundant schedules. For example, a trace-aware algorithm using the dependency relation from Definition 2.1 will not eliminate any reorderings of co-enabled events on the same node. However, this notion of dependency can be overly coarse: two co-enabled events at the same

node may be semantically commutative, that is, their reorderings always result in the same program behavior. Our solution is to perform a simple static analysis to identify commutativity.

Preliminary Dynamic Analysis. We instrument the code of a distributed system to expose the set of messages to the testing algorithm. Then, we estimate the set of racy events by running the system a number of times (up to n visible events) and gathering all racy events from all executions. That is, we estimate the set of racy events by including every event such that there is *some* execution in which it was racy. Then, when running the testing algorithm, we may identify an event as racy even if it is not racy in that particular execution. The performance of the taPCT algorithm is dependent on the precision of the race detection. More precise information about the set of racy events enables taPCT to identify more independent events and hence to eliminate a higher number of event reorderings. We make an assumption that the set of racy events collected by the above dynamic analysis approximates the set of possible racy events in the system.

While we use dynamic information to collect the set of events and racy events, one could also use static analysis to obtain an overapproximation of these sets. We opted for a dynamic implementation because of the complexity of scaling static analyses to large distributed systems.

Preliminary Static Analysis. We use a simple static analysis in order to refine the dependency relation by identifying commutative operations and marking them independent.

Consider a common case where a distributed system node receives a client request to write some value to a variable and then communicates the request to the other system nodes. Upon receiving the write request by this node, each node replies with a write-response message. A response message contains an acknowledgement or negative acknowledgement depending on the sender's local state. If the initiator node receives enough number of acknowledgement messages, it commits the client update to the database. The black-box definition of dependency relation labels the events of processing two write-response messages as dependent to each other. However, processing these messages only increment the count of acknowledgements and they are commutative. Given the black-box notion of dependency, trace-aware algorithms will not eliminate the reorderings of the commutative write-response messages. A more precisely defined dependence relation, which identifies two write-response messages as independent, reduces the number of traces explored.

We use a simple static analysis from [Leesatapornwongsa et al. 2014; Lukman et al. 2019] to identify simple cases of commutative events. These include event handlers that count write acknowledgements or leader election votes, or that write a constant value to a variable, or that discard an event. Such handlers are common in many distributed system protocols. While there may be more subtle reasons for commutativity, we only focus on these common cases for ease of implementation. We mark two events identified as commutative as independent, thereby refining the dependence relation. The marking of events is essentially a table of methods identified to be commutative, which is used by the implementation of the dynamic race checker when identifying racy events.

Note that the algorithms we compare can all use the coarser black-box notion of dependency in Definition 2.1, but employing the additional static analysis to refine the dependency relation increases the performance of trace-aware algorithms. In our evaluation below, we provide the same dependency relation to all algorithms when making comparisons, and we use static analysis only for the largest benchmark (Cassandra).

The rest of the paper is organized as follows. In Section 3, we take a brief theoretical detour to justify the theoretical guarantees for our algorithms. This section can be omitted on first reading. We formally present d−POS and taPCT in Section 4 and then perform an empirical comparison of these algorithms and their baselines in Section 5.

3 A THEORETICAL INTERLUDE: STRONGER PROBABILISTIC GUARANTEES?

Our randomized testing algorithms provide theoretical guarantees on the minimal probability of picking any trace graph. One might ask if there are algorithms with stronger guarantees, e.g., that sample trace graphs uniformly at random. Before proceeding to our main algorithms, we discuss a *static* algorithm with stronger guarantees and why it is insufficient in practice.

3.1 Almost Uniform Sampling from Traces

Let Σ be an alphabet of events and $D \subseteq \Sigma \times \Sigma$ the dependency relation. We consider the behavior of a distributed system as a set of schedules; this corresponds to a *language* over Σ. The dependence relation D induces an equivalence relation on schedules. Thus, given a language $L \subseteq \Sigma^*$ of schedules, we can define an associated language $\langle L \rangle_D$ of trace graphs: $\langle L \rangle_D = \{\langle w \rangle_D \mid w \in L\}$. Conversely, a trace graph $\langle w \rangle_D$ gives rise to a set of schedules obtained by any linearization of the graph.

We say a language $L \subseteq \Sigma^*$ of schedules is D-consistent if for all $w \in \Sigma^*$, $w \in L$ iff each schedule in $\langle w \rangle_D$ is in L. There is a 1-1 correspondence between languages of trace graphs and D-consistent languages over Σ. With this correspondence, a language of trace graphs is called *regular* iff it corresponds to a regular D-consistent language. (Recall that a regular language is one accepted by a finite-state automaton.)

Suppose we fix a bound n on the length of schedules, and consider the languages $L^{(n)}$ and $\langle L \rangle_D^{(n)}$ of schedules and trace graphs of size at most n. An ideal theoretical goal for a random testing algorithm would be to sample uniformly at random from the set $\langle L \rangle_D^{(n)}$. Unfortunately, complexity-theoretic considerations imply that if there was a uniform sampling procedure that ran in polynomial time, then problems in the complexity class #P would be solvable in polynomial-time.[3] Thus, one has to relax the requirement to *almost* uniform sampling.

A randomized algorithm A is an *almost uniform sampler* for $\langle L \rangle_D^{(n)}$ if given a description of $\langle L \rangle_D$, the bound n in unary, and $\epsilon > 0$, the algorithm $\mathsf{A}(\langle L \rangle_D, n, \epsilon)$ outputs some $y \in \langle L \rangle_D^{(n)}$ (if it exists) with probability at least $\frac{3}{4}$, and moreover, for each $y' \in \langle L \rangle_D^{(n)}$, we have

$$\frac{1}{|\langle L \rangle_D^{(n)}|(1+\epsilon)} \leq \Pr[\mathsf{A}(\langle L \rangle_D, n, \epsilon) = y'] \leq \frac{1+\epsilon}{|\langle L \rangle_D^{(n)}|} \tag{2}$$

Here, $|\langle L \rangle_D^{(n)}|$ is the cardinality of the language, i.e., the number of trace graphs of size at most n in $\langle L \rangle_D$. Informally, an almost uniform sampler produces outputs "close" to uniform, with a variability bounded by a factor of ϵ, and produces some output with high probability. When $\epsilon = 0$, the sampler A is a uniform sampler.

There is obviously a uniform sampler that runs in exponential time. Given a regular language of trace graphs, the question of whether there is an almost uniform sampler that runs in sub-exponential time has been open [Diekert and Rozenberg 1995; Sen 2007]. We show the following result.

Theorem 3.1. *Let Σ be a fixed alphabet and $D \subseteq \Sigma \times \Sigma$ a dependency relation. Let L be a regular trace language. If L is represented by an NFA A and n is in unary, then there is an almost uniform sampler for $\langle L \rangle_D^{(n)}$ that runs in time polynomial in $|A|$, n, and $\frac{1}{\epsilon}$ and exponential in $|\Sigma|$. If L is represented by a DFA A then there is a uniform sampler for $\langle L \rangle_D^{(n)}$ that runs in time polynomial in $|A|$ and n and exponential in $|\Sigma|$.*

[3] The complexity class #P [Arora and Barak 2009] consists of functions that compute the number of accepting computations of a non-deterministic polynomial-time Turing machine. The class #P contains (function) NP, and it is considered unlikely that problems in #P can be computed in polynomial time. Among others, it would imply P = NP.

The proof of this result depends on a normal form for trace graphs called the Foata normal form [Cartier and Foata 1969; Diekert and Rozenberg 1995], regularity properties of an automaton accepting Foata normal forms, as well as a polynomial time almost uniform sampling procedure for regular languages over words with a fixed alphabet [Arenas et al. 2019].[4]

3.2 Why is Theorem 3.1 Insufficient?

It may seem that Theorem 3.1 paves the way for an optimal random testing algorithm that picks each trace almost uniformly. Unfortunately, this is not the case.

First, the run time in Theorem 3.1 grows with the size of the size of the automaton. This automaton is, in general, *exponential* in the size of the program description. For example, for the distributed systems modeled in Section 2, each state of the automaton maps each node to its set of enabled events and the size of the automaton grows exponentially with the number of nodes. If we could construct the automaton, we could check if the automaton reaches an error state in linear time.

Second, in many systems, the static structure of the program may not be available, and the testing algorithm has to operate *online*: the algorithm has access to the current set of enabled events, and sees new events arising from executing an event only after executing the event. A natural question is whether one can still provide a strong guarantee similar to Theorem 3.1. Unfortunately, the following example shows that the online algorithm can perform exponentially worse.

Consider the following family of programs over the alphabet $\Sigma = \{a_1, \ldots, a_n\} \cup \{\hat{a}_1, \ldots, \hat{a}_n\}$ and the dependency relation $D = \{(a, a) \mid a \in \Sigma\} \cup \{(a_i, \hat{a}_i \mid i \in \{1, \ldots, n\}\}$. That is, the letters a_i and $\hat{a}_i$ are dependent, but letters with distinct indices are pairwise independent. There are $2n$ processes. The process $i \in \{1, \ldots, n\}$ produces the letter a_i and stops. Each process $j \in \{n + 1, \ldots, 2n\}$ either picks $\hat{a}_{j-n}$ or ϵ.

In the offline version, each program in the family has exactly two traces. By relabeling the alphabet, we can assume each program to be of the form $a_1 \parallel a_2 \ldots \parallel a_n \parallel \hat{a}_1 \parallel \ldots \parallel \hat{a}_k$ (the processes $n + k + 1$ to $2n$ did not emit a letter). An offline testing algorithm, that knows the program, needs to schedule two traces: $\langle a_1, \ldots, a_n\rangle\langle\hat{a}_1, \ldots, \hat{a}_k\rangle$ and $\langle\hat{a}_1, \ldots, \hat{a}_k, a_{k+1}, \ldots, a_n\rangle\langle a_1, \ldots, a_k\rangle$ with probability $\frac{1}{2}$ each.

However, consider an adversary that first presents the processes $1, \ldots, n$ before revealing the output of processes $n + 1$ to $2n$. The scheduler must pick a schedule of the first n processes, but since it does not know which letters should be delayed, it must pick every subset of $a_1, \ldots, a_n$ with positive probability. Since there are 2^n subsets, at least one subset is picked with probability at most $\frac{1}{2^n}$. For this subset, the adversary picks the corresponding $\hat{a}$'s, forcing the scheduler to schedule the trace with exponentially small probability. Thus, an online scheduler can pick a trace with exponentially smaller probability.

This is why we reduce our ambition and use the guarantee from Equation 1.

4 ONLINE SAMPLING ALGORITHMS : taPCT AND d-POS

In this section, we present taPCT and its special case d−POS. The goal of taPCT is to bring together two orthogonal objectives in randomized testing:

(i) *Trace-awareness.* Taking the dependency relation between the events into consideration not to cover some traces redundantly.
(ii) *Depth-bounding.* Sampling from *depth* bounded set of schedules which covers all depth-d bugs.

[4]At the time of our submission to OOPSLA, the the best almost uniform sampler for regular languages was the quasi-polynomial time (in n) procedure by Gore et al. [1997]. Since then, the recent breakthrough by Arenas et al. [2019] shows a fully polynomial time uniform sampler for regular languages.

Algorithm 1: taPCT algorithm

```
Input: The set of racy events R, bound n̂, bug depth d
Data: chains // list of chains of events in ascending order of priorities, the first d − 1 positions are initially null
Data: priorityChangePt // list of d − 1 distinct integers, initialized randomly between [1, n̂]
Data: racyEvents // number of racy events encountered, initially 0
Data: schedule // the current execution (a list of events)

Procedure addEvent(e)
1     if e ∈ R then
2         racyEvents ← racyEvents + 1
3         e.label ← racyEvents
      else
4         e.label ← null
5     insert e into a chain using online chain partitioning
6     if a new chain is created then
7         insert the new chain into a random position between d and |chains| in chains

Procedure scheduleNext()
8     c ← getHighestPriority(chains) // get the chain with the highest priority with an enabled event
9     e ← nextEnabledEvent(c) // get the next enabled event in the chain c
      // check if we are at a priority change point
10    if e.label ! = null ∧ ∃i : priorityChangePt[i] = e.label then
          // update the priority of the current chain c
11        swap c and chains[i] // note that chains[i] = null
12        return scheduleNext() // select the next event in the chain with highest priority
13    schedule.append(e)
14    return e
```

The key to depth bounding in our algorithms is the notion of *strong hitting families* of schedules [Kulahcioglu Ozkan et al. 2018]. A strong d-hitting family of schedules covers all bugs of depth d, where a bug is characterized by the precise ordering of a d-tuple of events $(e_0, \ldots, e_{d-1})$. Intuitively, a bug of depth d is exposed when the d events $(e_0, \ldots, e_{d-1})$ are scheduled in that order, and each event occurs as late as possible in the schedule.

Definition 4.1 (Strong hitting). Let $d \geq 1$ be an integer and let E be a set of events. We say a schedule s *strongly hits* a d-tuple of events $(e_0, \ldots, e_{d-1})$ if scheduling an event $e \in E$ after some e_i such that $0 \leq i \leq d-1$ implies that e is causally dependent to some e_j such that $j \geq i$.

Definition 4.2 (Strong d-hitting family). A set of schedules is called a *strong d-hitting family* for E if, for every d-tuple of elements in E, there is a schedule in the set that strongly hits it.

The taPCT algorithm samples from strong d-hitting families of schedules, like the PCT , and builds the schedule *online*. Having generated a d-tuple of events $(e_0, \ldots, e_{d-1})$, it samples a schedule strongly hitting the tuple which schedules e_i as late as possible in the execution before $e_i, \ldots, e_{d-1}$.

4.1 The taPCT Algorithm

The taPCT algorithm modifies PCT (specifically, the version of Kulahcioglu Ozkan et al. [Kulahcioglu Ozkan et al. 2018]) to make decisions aware of the race relation between events while sampling from a strong d hitting family of schedules.

- *Depth-bounding*. The taPCT algorithm builds on top of PCT which samples a strong d-hitting family of schedules based on a chain partitioning of events.
- *Trace-awareness.* While PCT selects the events to be strongly hit from all possible events in the execution, taPCT selects them only from the *racy* events.

Similar to the PCT algorithm, the taPCT algorithm assumes that multiple runs of the system have roughly the same characteristics. While PCT assumes that all the runs have the same number of events n, taPCT assumes that all the runs have the same number of *racy* events $\hat{n}$. This number is used as a bound for the selection of selecting d events to be strongly hit, i.e., taPCT selects d events out of $\hat{n}$ racy events.

Additionally, taPCT expects the set of racy events that can be observed in an execution as input. For this, taPCT requires an initial analysis to determine an approximation of the set of racy events. In Section 4.1.1, we describe the taPCT algorithm. In Section 4.1.2, we explain how we compute an approximation of the set of racy events. Finally in Section 4.1.3, we provide the lower bound on the guaranteed probability for sampling a bug of depth d.

4.1.1 The Algorithm. The taPCT algorithm represents the partial ordering of events in an execution internally as a set of *chains*, where each chain is a linearly ordered (causally dependent) sequence of events. Instead of assigning priorities to single events, as in POS, taPCT assigns priorities to chains. During execution, the algorithm picks the next enabled event from the highest priority chain and executes it. In addition, the algorithm can randomly change the priority of a chain at $d-1$ random points in the execution. The key difference from PCT is that the choice of priority change points is determined by looking only at racy events.

Algorithm 1 describes the taPCT algorithm, highlighting the modifications to PCT (from [Kulahcioglu Ozkan et al. 2018]) in blue. The algorithm takes three inputs, the approximation of the set of possibly racy events R, a number $\hat{n}$, and the depth bound d.

Like PCT, taPCT maintains three data structures: a list of chains of events (called *chains*) used to represent the partial order of events seen online during the execution, a list *priorityChangePt* of $d-1$ distinct integers used to randomly pick $d-1$ points along the execution, and a list *schedule* of events giving the current schedule. Each chain is assigned a priority; the list *chains* keeps the chains in ascending order w.r.t. their priorities, where the first $d-1$ slots with lowest priorities are initially empty. The list of $d-1$ integers in *priorityChangePt* are initialized randomly to be distinct integers in 1 to $\hat{n}$. They are used to identify at which event the priorities of the chains will be updated so that the schedule strongly hits the identified $d-1$ events in a particular order.

The taPCT algorithm has two procedures: addNewEvent inserts a new event into the chain partitioning and scheduleNewEvent picks the next event to be executed in the schedule. An outer loop (not shown in Algorithm 1) iteratively calls scheduleNewEvent, then executes the returned event, and calls addNewEvent with all the new events arising from the execution. The outer loop runs until some pre-specified number n of events have been executed.

When an event is added into the execution, the algorithm checks if it is a racy event or not. If it is a racy event, it is labeled with the current number of racy events encountered (lines 1-3). It is not labeled otherwise. Then, the event is inserted into an existing chain or into a new chain using the chain partitioning algorithm (lines 5-7). A newly created chain is randomly inserted into the list *chains*, reflecting a random priority among chains.

The next event to be scheduled is selected in `scheduleNext`. It finds the chain with the highest priority having an enabled event (line 8) and gets the next enabled event e in this chain (line 9). Then, it checks whether the priority of this chain will be reduced by checking whether the event is labeled and the label is identified as a priority change point (line 10). If so, it updates the priority of the chain containing e by relocating it in *chains* to an index in one of the first $d-1$ positions, corresponding to the index of e's label in *priorityChangePt* (line 11). Hence, the produced schedule strongly hits e preserving its label's order w.r.t. the other labels specified by *priorityChangePt*. If the priority of the current chain is updated, it selects the next event with the highest priority

by recursively calling `scheduleNext`. The algorithm terminates since the priority of chains are updated at only $d-1$ times in the execution identified by $d-1$ labels in *priorityChangePt*.

Like PCT, taPCT uses an online chain partitioning algorithm is used to decompose the partial order of events into a chain partitioning. The performance of taPCT depends on the underlying chain partitioning algorithm. In an extreme case, all events may be inserted into a different chain, we shall show later that this version of the algorithm captures d−POS. Theoretically, if the partial order is known statically and has width w (recall that the width of a partial order is the size of the largest antichain), there is a chain partitioning into w chains. However, an online algorithm may not find this partitioning. The key improvement is in the use of an online chain partitioning algorithm [Agarwal and Garg 2007] which guarantees that it builds at most $O(w^2)$ chains, where w is the width of the partial order.

The taPCT algorithm differs from PCT in the way that it selects the $d-1$ priority change points. While PCT does not consider the dependency relation while selecting the events to be strongly hit and selects $d-1$ events from all n events, taPCT selects the $d-1$ priority change points from racy events only, and indexes the priority change points using the interval $[1, \hat{n}]$ instead of $[1, n]$.

4.1.2 Computing the Set of Racy Events. The taPCT algorithm requires information about the set of racy events in the execution. Recall that two events are *racy* in an execution if they are *dependent* (causally, or because they are executed at the same node) and *may* be co-enabled in the execution. While we can use Definition 2.1 as a notion of dependency, it is often coarse. Moreover, we cannot judge if an event e is racy online, e.g., in the procedure addNewEvent or while scheduling e in the procedure scheduleNext. This is because an event dependent to e and co-enabled with it may not have been produced at the point we process e. Second, the scalability of taPCT depends on the precision of the dependency relation: the more events are identified to be independent, the fewer schedules are relevant. Therefore, the taPCT algorithm uses a preliminary analysis to determine an approximation of the set of racy events. We now explain our method for determining potentially racy events, i.e, whether an event is (i) dependent to, and (ii) co-enabled with, another event. As discussed in Section 2.5, we use a static and a dynamic analysis, respectively, for these steps.

(i) *Checking Dependency.* As is the case for all trace-aware algorithms, the taPCT algorithm is parametric w.r.t. the independence relation, where the precision of the independence relation increases the performance of taPCT. The dependence between two co-enabled events can be calculated by following Definition 2.1, by checking whether the two events will be processed on the same distributed system node. Although this generic and black-box notion is applicable to all distributed systems, it may not scale well for large real-word systems with many events. This is because Definition 2.1 labels a large number of events to be dependent, even though two events being processed on the same node may be commutative and hence be independent to each other. Therefore, in our implementation, we refine the dependency relation by employing the independence information extracted by the static analysis in [Leesatapornwongsa et al. 2014; Lukman et al. 2019].
Essentially, the static analysis provides the information of whether the execution of two events result in the same state. It runs in several phases. The first phase extracts the sets of variables read and updated by each event. If the read and update set of an event does not intersect with the read and update set of another event, the two events are labeled independent. In a second phase, the analysis also marks events as independent if they are semantically commutative according to a set of simple heuristics. For example, if the update sets of the two events are the same (i.e., they write to the same variables) and the updates are increment operations. We refer the reader to [Lukman et al. 2019] for more details of the static analysis.

For application where static information is difficult to obtain, the taPCT algorithm falls back to the coarser Definition 2.1.

(ii) *Collecting Racy Events.* The taPCT algorithm expects a set R of racy events as an input and uses it to pick a schedule. In our implementation, we use a dynamic analysis to approximate the set R of racy events, using the dependency relation obtained as above.
We run the system multiple times both to estimate a bound on the number of racy events in an execution, namely $\hat{n}$, and also to collect all racy events encountered (we identify each event by its sender, receiver and the content of the protocol message). Following the assumption on having the same characteristics in all executions of the system, we assume that the set of events so collected is representative of new runs. Note that the set is neither an overapproximation nor an underapproximation of the set of racy events. This is because we add an event to R if there exists some execution where it was racy, even though it may not race in a specific execution. On the other hand, we do not compute all possible executions of the system, and may miss a racy event.

4.1.3 The Probability of Sampling a Bug of Depth d using taPCT *Algorithm.* taPCT samples a schedule which strongly hits d events $(e_0, \ldots e_{d-1})$. It strongly hits $d-1$ events $(e_1, \ldots e_{d-1})$ selected among $\hat{n}$ racy events by updating their priorities to values smaller than 0. Additionally, it strongly hits an event e_0, which is in chain with the smallest priority. The tuple of $d-1$ events can be chosen in $\binom{\hat{n}}{d-1}(d-1)!$ different ways. Hence, the size of the set of schedules sampled by the taPCT algorithm is bounded by $numChains \times \binom{\hat{n}}{d-1}(d-1)! \leq numChains \times (\hat{n})^{d-1}$. Accordingly, the probability of sampling a bug of depth d is at least $1/(numChains \times (\hat{n})^{d-1})$.

4.2 The d−POS Algorithm

4.2.1 The Algorithm. The special case of taPCT where the chain partitioning algorithm puts each event into a new chain gives rise to the d−POS algorithm. It also turns out that this d−POS algorithm is also the natural modification of the POS algorithm [Yuan et al. 2018], restricted to sample a strong d hitting family of schedules for a given depth d.

The d−POS algorithm differs from POS in its priority updating scheme. d−POS updates the priorities of only $d-1$ racy events in a way that the produced schedule strongly hit them using *priorityChangeMap*. Different from POS which immediately schedules the event with the highest priority and updating the events of the priorities which are racy to the scheduled event, d−POS delays the execution of racy $d-1$ events to certain points in execution. As d increases to n, the total number of events, d−POS becomes the same as POS.

4.2.2 The Probability of Sampling a Bug of Depth d using $d-$POS Algorithm. The correctness argument for d−POS follows in the same way as for taPCT. However, the probabilistic guarantee uses the maximum concurrency rather than the number of chains. Let *maxConc* be the maximum number of events concurrently enabled with an event in the system. The chain partitioning strategy of d−POS gives rise to *maxConc* chains. Using the previous analysis, this means that the probability of sampling a bug of depth d is at least $1/(maxConc \times (\hat{n})^{d-1})$.

Note that the POS algorithm samples a schedule with a probability exponentially low in the number of events in the execution n; by biasing the search to schedules of low depth, d−POS provides a probability bound of hitting a bug that is exponentially small only in the depth parameter d. The experimental results in Section 5 shows the improvement of d−POS over POS in practice.

In general, $numChains \leq maxConc$, so theoretically, taPCT has an edge. The experimental results in Section 5 show the performance of taPCT over d−POS and PCT in terms of probability of

detecting bugs in practice, where this edge is borne out. Moreover, the correspondence with POS enables us to make precise statements about the empirical performance of taPCT against POS.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the bug finding performance of the POS, d−POS, PCT, and taPCT algorithms. We also compare against baseline random walk and RAPOS [Sen 2007] algorithms.

We briefly recall RAPOS. RAPOS improves random walk by scheduling a set of independent events together instead of selecting a single event to be scheduled. Thus, it avoids unnecessary sampling of an ordering between the selected independent events. RAPOS selects the set of events to be scheduled together in a randomized fashion. It starts with a random event and adds a subset of the enabled events to the set of scheduled events. That is, at any step, it adds an event into the subset with probability $\frac{1}{2}$ if that event is independent to the other events already in the subset. Then, it executes the selected subset of independent events without any particular order between them. RAPOS does not provide theoretical guarantees on the probability of sampling a trace.

We implement the algorithms on actor-based message passing systems in Akka and two large-scale systems Zookeeper and Cassandra. The Akka implementation is meant to evaluate the performance of the algorithms on a range of parameters, such as the chain length of causally dependent events as well as the number of racy and independent events on a set of micro-benchmarks.

In Section 5.1, we evaluate the effect of varying execution parameters (length of causally related chains of events and the ratio of the non-racy events) on the performance of the algorithms. Then, in Sections 5.2 and 5.3, we test two real world distributed systems Zookeeper and Cassandra, respectively. In the experiments, we compare the empirical distribution of sampled traces by each algorithm as well as their efficacy in finding bugs. While successfully reproducing the known bugs in these systems, our algorithms also found new inconsistencies in the most recent version of Zookeeper, v3.4.13.

Implementation of the algorithms. The major challenge in the application of the algorithms to different distributed systems is to intercept the events in the execution. This requires internal information about the implementation of the system under test and instrumentation of the code using AspectJ. Once the events are collected, the POS, d−POS, PCT, and taPCT algorithms schedule the intercepted events.

The trace-aware algorithms RAPOS, POS, d−POS and taPCT checks dependency among events while scheduling the events. In our evaluation, we use the notion of dependency in Definition 2.1 for Zookeeper which simply checks whether the receiver node of the two events are the same. However, the number of events in an execution of Cassandra is large, resulting in a huge number of traces. For this benchmark, we strengthened notion of dependency using the independence of events identified by the static analysis in [Leesatapornwongsa et al. 2014; Lukman et al. 2019]. We used the refined dependency relation for all trace-aware algorithms. This reduces the number of traces as we label some pairs of events as independent even if they are processed in the same node. We give concrete examples for independent events in Cassandra in Section 5.3. We compute an approximation to the set of racy events as described in Section 4.1.

5.1 Micro-benchmark Application in Akka Framework

We implemented the algorithms[5] together with a parametrized benchmark application[6] in Akka, a JVM based framework for actor based message passing programs [Agha and Hewitt 1985; Hewitt

[5]The source code of the algorithms is available at http://gitlab.mpi-sws.org/burcu/actor-pct
[6]The source code of the micro-benchmark is available at http://gitlab.mpi-sws.org/burcu/akkamicrobench

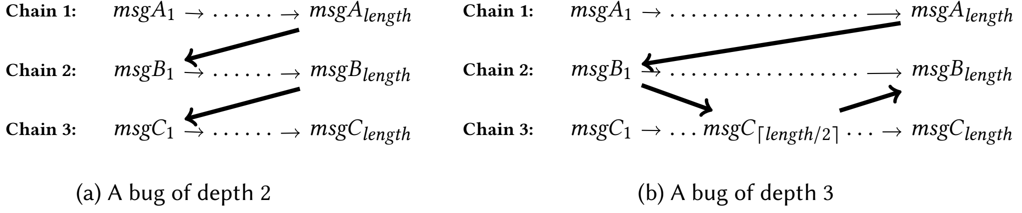


Fig. 4. Two bugs seeded into the micro-benchmark application. The bugs expose respectively when the events ($msgB_1$, $msgC_1$) and ($msgB_1$, $msgC_{\lceil length/2 \rceil}$, $msgC_{length}$) are strongly hit.

et al. 1973], to evaluate the effect of varying execution parameters on the algorithms. The micro-benchmark application is parameterized in the number of client requests, each of which create a chain of events to be processed, as well as in the length of the event chains and in the number of racy vs. non-racy events. For simplicity, we included the information of whether a message is racy in the content of the message. Thus, we did not require an initial analysis phase to identify racy events. We seeded two bugs of depth $d = 2$ and $d = 3$, respectively, into the application.

Figure 4 illustrates the seeded bugs on the benchmark program. The benchmark program starts with a number of concurrent requests each of which gives rise to a chain length *chainLength* of events to be processed. The program has *numChains* number of event chains, where some chains of events are racy with each other and some chains of events are independent. For simplicity, we only show the chains of racy events in the figure. Since both of the seeded bugs require 3 chains with racy events, we used at least $numChains = 3$ in our experiments. The scenario in the figure is instantiated by sending three messages $msgA_1$, $msgB_1$ and $msgC_1$ to the same node concurrently to each other. Depending on the benchmark configuration, the program may have more chains with racy events or some number of chains with independent events.

We designed the bugs so that they are exposed by nontrivial interleavings of a small number of events, following empirical bug patterns. The bug in Figure 4a is exposed when $msgA_{length}$ is executed before $msgB_1$ and also $msgB_{length}$ is executed before $msgC_1$. The bug is of depth $d = 2$ since it can be hit by strongly hitting the 2-tuple of events ($msgB_1$, $msgC_1$). However, the bug is not easy to hit by a naive random algorithm since it requires to schedule the events in the same chain successively before scheduling the first event in another chain. This bug is a variant of the example in the Example 2.2 in Section 2 with the difference that all the events in the chain are racy with an event in another chain.

The second bug in Figure 4b is exposed by a more complex ordering of events. In addition to sequencing all events in a chain before the first event in another chain, it also requires two chains of events to be interleaved. Specifically, the bug is hit by a schedule which sequences $msgA_{length}$ before $msgB_1$, $msgB_1$ before $msgC_{\lceil length/2 \rceil}$, and $msgC_{\lceil length/2 \rceil}$ before $msgB_{length}$. The bug is of depth $d = 3$ since it is exposed in a schedule which strongly hits the 3-tuple of events ($msgB_1, C_{\lceil length/2 \rceil}$, $msgB_{length}$). This bug is a variant of the example presented in Example 2.3 in Section 2 which exposes in a schedule that interleaves the chains of events.[7]

We evaluate the effect of using chain partitioning and trace awareness on the performance of the algorithms by varying some execution parameters. We run the algorithms on the benchmarks with varying (i) length of the causally related chains of events (chain length) and (ii) ratio of non-racy

[7]We experimented with variants of the bug in Figure 4b where we vary the position of the event in Chain 3 to be strongly hit. The experimental results on the variants show a similar trend in the frequency of detected bugs for varying benchmark parameters. We only show an average case where the bug requires the event in position $\lceil length/2 \rceil$ to be strongly hit.

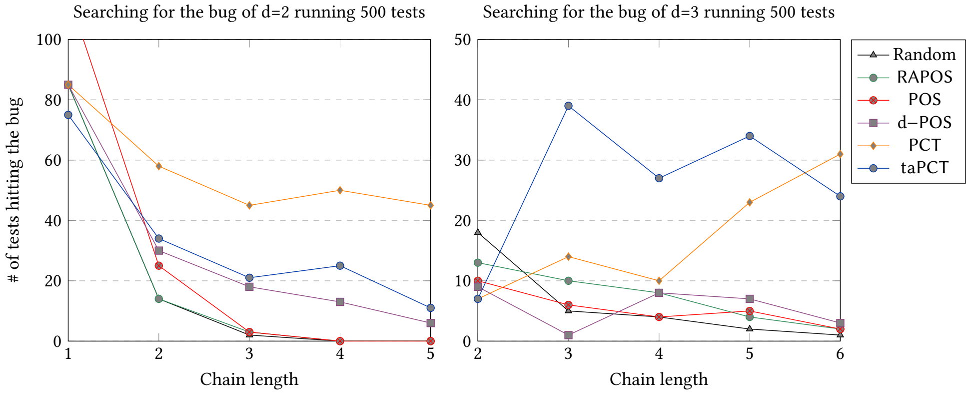


Fig. 5. The number of tests detecting the bugs for increasing chain length, for a fixed number of chains. Both of the bugs are seeded into programs having at least 3 chains with racy events. In the experiments, we used 3 chains with racy events and 3 chains with independent events all having the same chain length.

events to all events in the execution ($\alpha = 1 - (\hat{n}/n)$) which show the effect of chain partitioning and trace-awareness, respectively. For each parameter, we test the benchmark program 500 times using each of the algorithms and we report the total number of times the bug is found out of 500 tests (this is proportional to the empirical mean of finding the bug).

Varying chain length. The plots in Figure 5 show the number of tests hitting the bugs of depth $d = 2$ and $d = 3$ respectively for varying chain length. In both sets of experiments, we used 3 chains with racy events and 3 chains for non-racy events, where the ratio of non-racy events $\alpha = 1/2$. In the first set of experiments where we search for a bug of depth $d = 2$, we evaluated the algorithms for varying chain lengths in $[1, 5]$. In the second set of experiments, we varied the chain lengths in $[2, 6]$, since the bug of depth $d = 3$ requires at least 2 events to be in the same chain.

As the chain length increases, the difference in the performances of the algorithms operating on single events, i.e., random walk, POS and d−POS and the algorithms operating on chain partitions of events PCT and taPCT becomes visible. If the chain length is 1, then all messages are concurrent to each other. In this case, the performance of all algorithms approach a simple random walk. Notice that random walk, RAPOS and POS algorithms hit the bug with exponentially low probability in *chainLength*. These algorithms must repeatedly select the event from the same chain in order to hit the bug. The performance of d−POS algorithm drops since the increase in chain length also increases *maxConc*, the maximum number of events with which an event can be concurrently enabled. On the other hand, chain partitioning based algorithms have higher probability of hitting the bug as they successively schedule an event from the same chain until the priority of the chain is updated.

As the plots in Figure 5 show, all the algorithms detect the bug of depth $d = 3$ less frequently than the bugs of depth $d = 2$. This is expected, as the probability of hitting a bug of depth $d = 3$ is lower than hitting a bug of depth $d = 2$ for all algorithms. Different from the other algorithms, the performances of PCT and taPCT are not directly affected by the chain length, but by the total number of events n. The plots have spikes at the points where PCT or taPCT algorithms detect the bugs with higher frequency than might be expected. These can be explained by the fact that these bugs can be exposed by multiple different schedules which are likely to be produced by chain partitioning based algorithms. Consider the seeded bug of depth $d = 2$ that only requires sequencing some chains of events without any interleavings. No matter which events are strongly

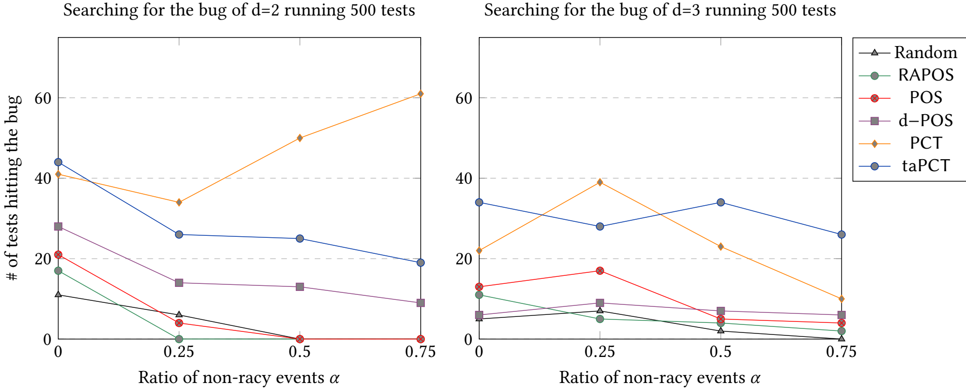


Fig. 6. The number of tests detecting the bugs for increasing α. For the experiments for searching for the bugs of $d = 2$ and $d = 3$, the chain length is 2 and 3 respectively. In both experiments, we used 3 chains with racy events and increasing number of chains with non-racy events for different α values.

hit (scheduled latest), a permutation of chains produced by PCT or taPCT will hit the bug provided that the permutation meets the relative order exposing the bug. Similarly, the seeded bug of depth $d = 3$ is not only exposed by strongly hitting $(msgB_1, C_{\lceil length/2 \rceil}, msgB_{length})$ but also by several other tuples. Specifically, consider a schedule strongly hitting any event e (other than $msgC_1$) in Chain 3 that appears earlier than $msgC_{\lceil length/2 \rceil}$ as the second event in the tuple. Such a schedule executes $msgC_{\lceil length/2 \rceil}$ between the events $msgB_1$ and $msgB_{length}$ and hits the bug.

Varying ratio of non-racy events to all events: α. The plots in Figure 6 show the number of tests hitting the bugs of depth $d = 2$ and $d = 3$ respectively, with increasing ratio of non-racy events in the execution. In both sets of experiments, the executions have 3 chains with racy events and the chain length is 2 for all the chains. We experiment with varying ratio of non-racy events $\{0, 0.25, 0.5, 0.75\}$ by increasing the number of chains with non-racy events as necessary. In one extreme, we have $\alpha = 0$ which corresponds to the case where all events are racy. On the other extreme with $\alpha = 1$, there are no racy events, i.e., all possible executions are equivalent w.r.t. the dependency relation D.

As the ratio of non-racy events increases, the bug detecting performance of an algorithm that is not trace-aware degrades. This is more visible for the more complex bug with depth $d = 3$. The performance of taPCT algorithm does not significantly change for the increasing ratio of non-racy events. However random walk, RAPOS, and POS detect fewer bugs with increasing α. Despite being trace-aware, the performance of d−POS degrades with increasing ratio of non-racy events as well. This is due to the indirect increase in *maxConc* as we increase the number of independent events in the program. The random spikes in the frequency of the bugs detected by the PCT algorithm can be explained similarly to the case in Figure 5.

The experiments searching for the bugs of depth $d = 3$ in Figure 6 highlight the performance difference between PCT and taPCT algorithms, whose probabilistic guarantees only differ w.r.t. α. On the plot for the bug of depth $d = 2$ in Figure 6, the algorithms select only 1 priority change point and they have similar performance. However, the improvement of taPCT over PCT is more significant for higher values of bug depth d. This is because PCT selects $d - 1$ priority change points from n events whereas taPCT selects them from $(1 - \alpha)n = \hat{n}$ events.

Table 1. Results for testing Zookeeper v3.4.3 - Sampling from all executions

Sampler	#class	min	max	mean	dev	#faulty
random walk	853	1	8	1.17	0.63	5
RAPOS	560	1	19	1.79	1.56	4
POS	952	1	5	1.05	0.27	16

5.2 Zookeeper

Zookeeper[8] is a distributed key-value store used by large distributed systems for maintaining configuration and naming information, providing distributed synchronization, and for other purposes that arise while coordinating nodes in a distributed setting. Its intended usage requires Zookeeper to provide strong consistency guarantees that is achieved by running a distributed consensus algorithm called *Zab* (Zookeeper Atomic Broadcast) [Junqueira et al. 2011].

In our evaluation[9], we run Zookeeper with 3 nodes. We consider the leader election protocol executions with the following inconsistencies as faulty when the algorithm terminates and announces that a leader has been elected:

(1) More than one (alive) node thinks they are leaders
(2) There are some (alive) nodes that are neither leading nor following a leader
(3) There are some (alive) nodes that follow a node other than the reported leader node

Our evaluation for the Zookeeper injects a number of node crash and node reboot events in the execution. We add a node crash event causally dependent on the node start event for each node. Similarly, for each node crash event, we introduce a node reboot event causally dependent to the crash event. The crash and reboot events are enabled if the executed number of node crash and reboot events are smaller than a user provided parameter for the maximum number of node crashes and reboots respectively. In our tests, we used a budget of 1 node crash event and 1 node reboot event, that is, we consider executions in which up to one node can crash and then recover.

In the experiments we used two versions of Zookeeper v3.4.3 and v3.4.13 (the newest version at the time of writing this paper). We tested both versions by running each testing algorithm for 1000 executions.

We analyzed the performance of the algorithms in two aspects. First, we measure the empirical uniformity of sampling. In Tables 1–3, the columns *#class*, *min*, *max*, *mean* and *dev* (standard deviation) measure uniformity of sampling. The *#class* column summarizes the number of different trace graphs hit during testing. We bucket the tests into these trace graphs. A fully uniform sampler would pick each bucket the same number of times, in expectation, and the standard deviation will be low. The columns max and min report the maximum and minimum number of tests that went into a bucket; *mean* is the average number of tests in a bucket and *dev* is the standard deviation. Second, we measure the number of faulty executions falling to the inconsistent cases (1), (2), and (3) above. We list them in the columns *type*1, *type*2 and *type*3 respectively together with their summation in the column *#faulty* in Tables 1–3).

Table 1 shows the experimental results for testing Zookeeper v3.4.3 using random walk, RAPOS, and POS algorithms. The executions sampled by an algorithm is closer to uniform distribution if all traces are sampled with an equally likely probability. As the table shows, random walk and RAPOS

[8] http://zookeeper.apache.org
[9] The source code is available at http://gitlab.mpi-sws.org/rupak/hitmc/tree/develop/HitMC

Table 2. Results for testing Zookeeper v3.4.3 - Sampling from strong d-hitting families of schedules

Sampler	d	#class	min	max	mean	stddev	#faulty	type 1	type 2	type 3
PCT	4	470	1	47	2.13	2.96	35	27	2	7
	5	552	1	39	1.81	2.40	26	26	0	0
	6	636	1	25	1.57	1.75	31	25	2	4
d−POS	4	659	1	37	1.52	1.88	12	10	0	2
	5	597	1	45	1.68	2.38	18	18	0	0
	6	537	1	48	1.86	2.83	16	13	0	3
taPCT	4	329	1	76	3.04	4.89	40	38	0	2
	5	310	1	96	3.23	5.10	33	27	0	6
	6	317	1	75	3.15	5.06	34	28	0	6

Table 3. Results for testing Zookeeper v3.4.13 - Sampling from strong 4-hitting families of schedules

Sampler	#class	min	max	mean	stddev	#faulty	type 1	type 2	type 3
PCT	479	1	45	2.09	2.97	43	24	11	8
d−POS	647	1	33	1.55	1.87	20	9	9	2
taPCT	313	1	78	3.19	5.05	45	32	11	2

algorithms do not sample uniformly. The big difference in the empirical minimum and maximum number of times of sampling a particular trace, and high value of standard deviation shows that these algorithms sample different traces with significantly varying frequencies. The distribution of sampled traces by the POS algorithm is closer to uniform, where the sampling frequencies of traces do not deviate as much as the other algorithms. All these algorithms find some inconsistencies.

Table 2 shows the experimental results using PCT, d−POS, and taPCT algorithms for different values of bug depth. As shown on the table, these algorithms find faulty executions more frequently than the algorithms in Table 1. This is because PCT, d−POS, and taPCT sample from the bounded set of strong d-hitting family of schedules while RAPOS and POS sample from the set of all possible executions. Bounding the sample set to strong d-hitting family of schedules increases the probability of finding depth d bugs. While we do not observe significant differences between PCT and d−POS in the uniformity of the samples and the frequency of detecting faults in our evaluation, we observe that taPCT mostly outperforms the other algorithms in the detected number of faulty executions. taPCT is slightly "less uniform", and this may be explained by its scheduling all enabled events in a chain before moving to the next chain. However, even though it covers fewer traces than the other algorithms, it is surprisingly effective in finding bugs, compared to the other techniques.

In all three algorithms, the number of faults detected using $d = 4$ is higher than the tests using higher depth values. This can be explained by the fact that the guaranteed probability of hitting a bug of depth d decreases when we sample from a larger sample set, i.e. strong hitting families of schedules with higher d values. Table 3 compares PCT, d−POS, and taPCT in the newest version of Zookeeper v3.4.13 with bug depth $d = 4$. For these experiments, all algorithms perform sufficiently close to each other, both in terms of uniformity of sampling and in the number of inconsistencies detected.

5.3 Cassandra

Cassandra[10] is a distributed NoSQL database management system which provides lightweight transactions based on the Paxos consensus protocol. We tested Cassandra version 2.0.0, which is known to have a deep concurrency bug[11] in its Paxos protocol implementation. Cassandra's implementation of the Paxos protocol has three phases. In the *prepare/promise* phase, the leader node that proposes a value to write picks a ballot number and sends it to the other nodes. If the ballot is the highest a receiving node has seen, it promises not to accept any proposals with an earlier ballot. If a majority of nodes promise to accept the proposal, the leader node continues with the *propose/accept* phase to accept the value. Cassandra extends Paxos protocol with a third *commit* phase that resets the Paxos state of the nodes for the next proposals.

The bug is exposed in a scenario where three Cassandra nodes process three client transactions to write some values to some keys concurrently. The nodes exchange the protocol messages for each transaction concurrently with each other. In a subtle interleaving, some protocol messages of some transactions arrive at some nodes after some messages pertaining to other transactions. The buggy interleaving causes to commit a transaction more than once, resulting in data inconsistency.

The bug is hard to detect as it requires subtle reorderings of several protocol messages in an execution of 54 events. If we use the notion of dependency given by Definition 2.1, each of the three nodes process 18 messages whose reorderings result in a huge number of possible non-equivalent executions. Assigning each unique ordering into a different equivalence class of executions leads to large number of equivalence classes many of which produce the same behavior.

To coarsen the partitioning into equivalence classes, we employed a static analysis described by Leesatapornwongsa et al. [2014] to detect commutativity of some events. The commutativity information allows us to identify two events as independent even when they are processed at the same node. Given a set of commutativity conditions provided by Leesatapornwongsa et al. [2014], we override the part of the testing algorithm which checks the dependency between the events. While the default method checks only whether two given events will be processed in the same node, we override this method to use the message contents of the events to decide whether they are commutative or not. Note that it is possible to use the default method for checking dependency. However, in order to refine the relation for a system under test, the programmer needs to override the dependency checking method using the commutativity conditions specific to the system.

We used commutativity of the following events in Cassandra's Paxos protocol implementation:

- Messages to be discarded:
  - A response to a *commit* message is discarded and it is independent with all other messages
- Messages whose handlers write to disjoint sets of variables:
  - A *prepare* message is independent with a response to a *prepare* message
  - A *propose* message is independent with a response to a *propose* message
  - A *commit* message is independent with a response to a *prepare* message
  - A response to a *prepare* message is independent with a response to a *propose* message
- The messages whose handlers only increment a variable value:
  - Two response messages to a *propose* message

Table 4 shows the results of testing Cassandra with random walk, RAPOS, and POS algorithms for 1000 executions.[12] All algorithms were given the same dependency relation (obtained by the static analysis). As with Zookeeper, the column #*class* shows the number of trace graphs sampled in the tests. The executions sampled by an algorithm is closer to uniform distribution if all traces

[10] http://cassandra.apache.org

[11] https://issues.apache.org/jira/browse/CASSANDRA-6023

[12] The source code is available at http://gitlab.mpi-sws.org/burcu/pctcp-cass/tree/taPCT

Table 4. Results for testing Cassandra v2.0.0 - Sampling from all executions

Sampler	#class	min	max	mean	dev	#buggy
random walk	992	1	4	1.01	0.12	0
RAPOS	994	1	2	1.01	0.08	0
POS	1000	1	1	1.00	0.00	0

Table 5. Results for testing Cassandra v2.0.0 - Sampling from strong d-hitting families of schedules

Sampler	d	#class	min	max	mean	dev	#buggy
d−POS	4	998	1	2	1.00	0.04	0
	5	1000	1	1	1.00	0.00	0
	6	1000	1	1	1.00	0.00	1
PCT	4	876	1	4	1.14	0.44	0
	5	904	1	4	1.11	0.37	1
	6	915	1	5	1.09	0.38	1
taPCT	4	877	1	6	1.14	0.47	2
	5	896	1	9	1.12	0.49	3
	6	903	1	7	1.11	0.41	3

are sampled with an equally likely probability. In our evaluation of 1000 tests, we do not observe significant difference in the uniformity of random walk, RAPOS, and POS algorithms, although POS performed slightly better. POS sampled each trace once as listed by the minimum and maximum number of times of sampling a trace and the standard deviation of the frequency of hitting a trace. However, none of the algorithms could find the bug within 1000 executions. This is not surprising since the possible number of executions of 54 events, even with the static analysis, result in a huge sampling space.

Depth bounded randomized testing algorithms avoid the huge sample space of possible executions by searching for the bugs in a bounded set of executions. We show the results of testing Cassandra on PCT, d−POS, and taPCT algorithms in Table 5. Different from the algorithms in Table 4, which sample from all possible executions and cannot not find the bug in 1000 tests, sampling from strong hitting schedules of events successfully expose the bug. While we do not observe significant differences in the uniformity of the samples, the algorithms differ in the number of times they detect the bug. The PCT algorithm detects the bug in one execution with each of $d = 5$ and $d = 6$, and d−POS algorithm can detect it in an execution with $d = 6$. The taPCT algorithm detects the bug most number of times for all $d = 4, 5, 6$.

In summary, there is not much distinction in empirical uniformity of sampling, but taPCT again outperforms the other techniques in terms of bug finding.

6 RELATED WORK

Systematic concurrency testing exercises all possible behaviors of a program, which suffers from state space explosion due to the exponentially high number of possible executions in the number of events in the program. Partial order reduction (POR) techniques [Abdulla et al. 2014, 2017; Flanagan and Godefroid 2005; Godefroid 1996] reduce the state space by pruning out equivalent executions.

While the traditional POR techniques use the equivalence notion defined by Mazurkiewicz traces, recent work aims to coarsen the equivalence by exploiting dependency relation in certain concurrency models [Tasharofi et al. 2012], considering a context-sensitive dependency relation based on whether the events are dependent in a certain context [Albert et al. 2017], or an observation equivalence based on the read values of the variables [Chalupa et al. 2018]. However, practical systems are too large for POR techniques to exhaustively explore all traces.

Bounded exploration approaches tackle the state space problem by focusing on a subset of possible executions which are likely to expose concurrency bugs. The idea builds on a notion of bug depth and parameterizing the search space by the depth d. Context bounding [Qadeer and Rehof 2005] characterizes the depth as the number of context switches between the threads in a multithreaded program. Preemption bounding [Coons et al. 2013; Musuvathi and Qadeer 2007] only bounds the preemptive context switches, leaving the non-preemptive switches (due to thread blocking or termination) unbounded. While context and preemption bounding are effective for the analysis of multithreaded systems, they are not as effective for asynchronous or distributed systems, where the number of tasks are unbounded. Delay bounding [Desai et al. 2015; Emmi et al. 2011] bounds only the number of deviations from the given deterministic scheduler, making the exploration cost independent in the number of concurrent tasks. Phase bounding [Bouajjani and Emmi 2012] defines depth as the number of process communication cycles in a distributed message passing system. Previous empirical work on bounded schedulers [Thomson et al. 2014] showed that the performance of a random scheduler competes with t he more sophisticated bounding approaches in detecting bugs. In this work, we focus on randomized algorithms and use a notion of bug depth (which is also used in previous work [Burckhardt et al. 2010; Chistikov et al. 2016]) characterized by the ordering constraints [Long et al. 2016; Lucia and Ceze 2009] between the events in a system.

Several tools, such as MaceMC [Killian et al. 2007], MODIST [Yang et al. 2009], dBug [Simsa et al. 2011], and testing for P# framework [Deligiannis et al. 2016] are proposed for systematic exploration of distributed systems. These mainly differ in the target distributed systems. Typically, these tools use a controlled scheduler to enforce a particular order of program events and reduce the state space by applying POR techniques or bounding the set of explored executions [Desai et al. 2015]. With a goal of detecting bugs due to the ordering of two events, Bita [Tasharofi et al. 2013] covers only the pairwise orderings of the program events. Recent work [Leesatapornwongsa et al. 2014; Liu et al. 2017; Lukman et al. 2019] reduces the state space by using some semantic information about the system under test. Different from the systematic testing approaches with a controlled scheduler, Falcon [Leners et al. 2011] and Jepsen [Kingsbury 2018] modify the environment in which the system under test runs and stress test the system to expose faulty behaviors. Unfortunately, we do not know how to provide probabilistic guarantees for these systems.

7 CONCLUSION

We presented online randomized algorithms to sample schedules from distributed message passing systems in a trace-aware and depth-bounded manner. The taPCT algorithm generalizes related algorithms from the testing literature, and unifies two orthogonal reductions of the search space. Our evaluations on micro-benchmarks as well as on large distributed applications show that taPCT can be effective in detecting inconsistencies in these systems. While we have presented taPCT for distributed message-passing systems, it is also applicable to multithreaded shared-memory systems. We leave its evaluation on such systems to future work.

A PROOF OF THEOREM 3.1

Theorem 3.1 shows an almost uniform sampler from the n-slice of a regular trace language. We prove Theorem 3.1 using two ingredients: a normal-form for traces and a recent polynomial-time almost-uniform sampling theorem for regular word languages.

Happens-before graphs can canonically be represented using *Foata normal form* [Cartier and Foata 1969; Diekert and Rozenberg 1995]. Let Σ be a fixed alphabet and $D \subseteq \Sigma \times \Sigma$ a dependence relation. The alphabet $Ind(\Sigma)$ of non-empty independent sets over Σ is defined as:

$$Ind(\Sigma) = \{\Sigma' \subseteq \Sigma \mid \Sigma' \neq \emptyset \text{ and } \forall a, b \in \Sigma' : (a, b) \notin D\}$$

Note that $|Ind(\Sigma)| \leq 2^{|\Sigma|}$.

We say the pair $(\sigma, \sigma') \in Ind(\Sigma) \times Ind(\Sigma)$ is *compatible* over (Σ, D) iff for all $a \in \sigma'$ there is $b \in \sigma$ and $(a, b) \in D$. Define the morphism $\pi : (Ind(\Sigma) \cup \{\emptyset\})^* \longrightarrow \mathcal{T}(\Sigma, D)$ as $\pi(\emptyset) = [\epsilon]_\equiv$ and $\pi(\{a_1, \ldots, a_k\}) = [a_1 a_2 \ldots a_k]_\equiv$. A word $t_1 \ldots t_n \in Ind(\Sigma)^*$ is *proper* if (t_i, t_{i+1}) is compatible for each $i \in \{1, \ldots, m-1\}$. The *Foata normal form* of a trace t over (Σ, D) is the unique proper word $t_1 t_2 \ldots t_m \in Ind(\Sigma)^*$ such that $\pi(t_1 t_2 \ldots t_m) = t$.

Intuitively, the Foata normal form of a trace schedules all minimal independent letters together; moreover, it schedules a minimal element as soon as possible. Given a happens-before graph $\langle x \rangle_D$, one can produce the Foata normal form by running the following algorithm: write out all minimal (and therefore independent) elements of the graph, delete them from the graph and continue with the rest of the graph until no nodes remain.

For a trace t, we write $fnf(t)$ to denote the unique Foata normal form of t. For a language L of traces, we write $fnf(L)$ to denote $\{fnf(t) \mid t \in L\}$.

Lemma A.1. *(1) [Cartier and Foata 1969; Diekert and Rozenberg 1995] Each trace has a unique Foata normal form.*
(2) The set of all proper words in $Ind(\Sigma)^*$ is a regular (word) language.
(3) Let L be a regular language of traces. Then $fnf(L)$ is a regular language over $Ind(\Sigma)^*$.

Proof. Part (1) is proved in [Diekert and Rozenberg 1995].

For part (2), we define a DFA $A_{fnf} = (Q, Ind(\Sigma), \delta, q_0, F)$ over $Ind(\Sigma)^*$ that recognizes all proper words:

- $Q = Ind(\Sigma) \cup \{q_0, \mathsf{err}\}$
- $\delta : Q \times Ind(\Sigma) \mapsto Q$ is a transition relation such that: $\delta(q_0, \sigma') = \sigma'$, $\delta(\sigma, \sigma') = \sigma'$ if $\sigma, \sigma' \in Ind(\Sigma)$ and (σ, σ') is compatible and $\delta(\sigma, \sigma') = \mathsf{err}$ if σ, σ' are not compatible, and $\delta(\mathsf{err}, \cdot) = \mathsf{err}$,
- q_0 is the initial state, and
- $F = Q \setminus \{\mathsf{err}\}$.

Intuitively, the state of the automaton remembers the last symbol seen and allows a transition only if the input symbol is compatible with the last symbol stored in the state. The size of A_{fnf} is $O(|Ind(\Sigma)|^2)$.

For part (3), we first show that $\pi^{-1}(L)$ is a regular language, using the result that regular languages are closed under inverse homomorphism [Hopcroft and Ullman 1979]. The language $\pi^{-1}(L)$ consists of words in $Ind(\Sigma)^*$, but the words may not be proper. To fix this, we intersect this language with $L(A_{fnf})$. The intersection consists of exactly the set of Foata normal forms of traces in L. □

The second ingredient is a breakthrough result by Arenas et al. [2019] on polynomial-time almost uniform sampling of regular word languages, which implies the following lemma.

Lemma A.2. *[Arenas et al. 2019] There is an almost uniform sampler to sample from $L(A)^{(n)}$ for an NFA A over a fixed alphabet Σ that runs in time polynomial in $|A|$, n, and $\frac{1}{\epsilon}$.*

Now, to prove the theorem, we use Lemma A.1 to construct a word language from the regular language of happens-before graphs: given A, we take the intersection of $L(A)$ with $L(A_{fnf})$. The resulting automaton has size $|A| \cdot |A_{fnf}|$. We then apply Lemma A.2. When A is a DFA, one can use a dynamic programming algorithm to count the number of accepted words in polynomial time.

One could ask if the exponential dependence on Σ can be removed. This is likely to be hard as it will imply a fully polynomial time approximation scheme to count (or sample) the number of acyclic orientations of a graph, a well known open problem.

ACKNOWLEDGMENTS

This research was funded in part by the Deutsche Forschungsgemeinschaft (DFG) grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science) and by the European Research Council Grant Agreement No. 610150 (ERC Synergy Grant ImPACT (http://www.impact-erc.eu/)). We thank Ritabrata Ray for useful discussions.

REFERENCES

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, 373–384.

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2017. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. *J. ACM* 64, 4 (2017), 25:1–25:49. https://doi.org/10.1145/3073408

Anurag Agarwal and Vijay K. Garg. 2007. Efficient dependency tracking for relevant events in concurrent systems. *Distributed Computing* 19, 3 (2007), 163–183. https://doi.org/10.1007/s00446-006-0004-y

Gul Agha and Carl Hewitt. 1985. Concurrent Programming Using Actors: Exploiting large-Scale Parallelism. In *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings.* 19–41. https://doi.org/10.1007/3-540-16042-6_2

Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. 2017. Context-Sensitive Dynamic Partial Order Reduction. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I.* 526–543. https://doi.org/10.1007/978-3-319-63387-9_26

Marcelo Arenas, Luis Alberto Croquevielle, Rajesh Jayaram, and Cristian Riveros. 2019. Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 59–73. https://doi.org/10.1145/3294052.3319704

Sanjeev Arora and Boaz Barak. 2009. *Computational Complexity - A Modern Approach.* Cambridge University Press.

Ahmed Bouajjani and Michael Emmi. 2012. Bounded Phase Analysis of Message-Passing Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings.* 451–465. https://doi.org/10.1007/978-3-642-28756-5_31

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010.* 167–178. https://doi.org/10.1145/1736020.1736040

P. Cartier and D. Foata. 1969. Problèmes combinatoires de commutation et r'earrangements. Number 85 in Lecture Notes in Mathematics. Springer.

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2018. Data-centric dynamic partial order reduction. *PACMPL* 2, POPL (2018), 31:1–31:30. https://doi.org/10.1145/3158119

Dmitry Chistikov, Rupak Majumdar, and Filip Niksic. 2016. Hitting Families of Schedules for Asynchronous Programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 9780. Springer, 157–176. https://doi.org/10.1007/978-3-319-41540-6_9

Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. 2013. Bounded partial-order reduction. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013.* 833–848. https://doi.org/10.1145/2509136.2509556

Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. 2016. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara,*

CA, USA, February 22-25, 2016. 249–262. https://www.usenix.org/conference/fast16/technical-sessions/presentation/deligiannis

Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. 2015. Systematic testing of asynchronous reactive systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015.* 73–83. https://doi.org/10.1145/2786805.2786861

Volker Diekert and Grzegorz Rozenberg (Eds.). 1995. *The Book of Traces.* World Scientific.

Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* 411–422. https://doi.org/10.1145/1926385.1926432

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05).* ACM, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315

Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 174–186.

Vivek Gore, Mark Jerrum, Sampath Kannan, Z. Sweedyk, and Stephen R. Mahaney. 1997. A Quasi-Polynomial-Time Algorithm for Sampling Words from a Context-Free Language. *Information and Computation* 134, 1 (1997), 59–74.

Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973.* 235–245. http://ijcai.org/Proceedings/73/Papers/027B.pdf

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation* (1st ed.). Addison-Wesley.

Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)* (2011), 245–256.

Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code (Awarded Best Paper). In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings.* http://www.usenix.org/events/nsdi07/tech/killian.html

Kyle Kingsbury. 2013–2018. *Jepsen.* Retrieved April 05, 2019 from http://jepsen.io/

Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *PACMPL* 2, OOPSLA (2018), 160:1–160:28.

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* 399–414. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa

Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. 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, GA, USA, April 2-6, 2016.* 517–530. https://doi.org/10.1145/2872362.2872374

Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos Kawazoe Aguilera, and Michael Walfish. 2011. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011.* 279–294. https://doi.org/10.1145/2043556.2043583

Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. 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 2017, Xi'an, China, April 8-12, 2017.* 677–691. https://doi.org/10.1145/3037697.3037735

Yuheng Long, Mehdi Bagherzadeh, Eric Lin, Ganesha Upadhyaya, and Hridesh Rajan. 2016. On ordering problems in message passing software. In *Proceedings of the 15th International Conference on Modularity, MODULARITY 2016, Málaga, Spain, March 14 - 18, 2016.* 54–65. https://doi.org/10.1145/2889443.2889444

Brandon Lucia and Luis Ceze. 2009. Finding concurrency bugs with context-aware communication graphs. In *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA.* 553–563. https://doi.org/10.1145/1669112.1669181

Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019.* 20:1–20:16. https://doi.org/10.1145/3302424.3303986

Rupak Majumdar and Filip Niksic. 2018. Why is random testing effective for partition tolerance bugs? *PACMPL* 2, POPL (2018), 46:1–46:24. https://doi.org/10.1145/3158134

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 446–455. https://doi.org/10.1145/1250734.1250785

Santosh Nagarakatte, Sebastian Burckhardt, Milo M. K. Martin, and Madanlal Musuvathi. 2012. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 543–554. https://doi.org/10.1145/2254064.2254128

Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. 93–107. https://doi.org/10.1007/978-3-540-31980-1_7

Koushik Sen. 2007. Effective random testing of concurrent programs. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. 323–332. https://doi.org/10.1145/1321631.1321679

Jirí Simsa, Randy Bryant, and Garth A. Gibson. 2011. dBug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. In *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*. 188–193. https://doi.org/10.1007/978-3-642-22306-8_14

Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. 2012. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*. 219–234. https://doi.org/10.1007/978-3-642-30793-5_14

Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph E. Johnson. 2013. Bita: Coverage-guided, automatic testing of actor programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 114–124. https://doi.org/10.1109/ASE.2013.6693072

Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2014. Concurrency testing using schedule bounding: an empirical study. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. 15–28. https://doi.org/10.1145/2555243.2555260

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. 213–228. http://www.usenix.org/events/nsdi09/tech/full_papers/yang/yang.pdf

Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. Partial Order Aware Concurrency Sampling. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. 317–335.