Partial Order Aware Concurrency Sampling

Xinhao Yuan(✉), Junfeng Yang(✉), and Ronghui Gu

Columbia University, New York, USA
{xinhaoyuan,junfeng,rgu}@cs.columbia.edu

Abstract. We present POS, a concurrency testing approach that samples the partial order of concurrent programs. POS uses a novel priority-based scheduling algorithm that dynamically reassigns priorities regarding the partial order information and formally ensures that each partial order will be explored with significant probability. POS is simple to implement and provides a probabilistic guarantee of error detection better than state-of-the-art sampling approaches. Evaluations show that POS is effective in covering the partial-order space of micro-benchmarks and finding concurrency bugs in real-world programs, such as Firefox's JavaScript engine SpiderMonkey.

1 Introduction

Concurrent programs are notoriously difficult to test. Executions of different threads can interleave arbitrarily, and any such interleaving may trigger unexpected errors and lead to serious production failures [13]. Traditional testing over concurrent programs relies on the system scheduler to interleave executions (or events) and is limited to detect bugs because some interleavings are repeatedly tested while missing many others.

Systematic testing [9,16,18,28–30], instead of relying on the system scheduler, utilizes formal methods to systematically schedule concurrent events and attempt to cover all possible interleavings. However, the interleaving space of concurrent programs is exponential to the execution length and often far exceeds the testing budget, leading to the so-called *state-space explosion* problem. Techniques such as partial order reduction (POR) [1,2,8,10] and dynamic interface reduction [11] have been introduced to reduce the interleaving space. But, in most cases, the reduced space of a complex concurrent program is still too large to test exhaustively. Moreover, systematic testing often uses a deterministic search algorithm (e.g., the depth-first search) that only slightly adjusts the interleaving at each iteration, e.g., flip the order of two events. Such a search may very well get stuck in a homogeneous interleaving subspace and waste the testing budget by exploring mostly equivalent interleavings.

To mitigate the state-space explosion problem, randomized scheduling algorithms are proposed to *sample*, rather than enumerating, the interleaving space

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10982, pp. 317–335, 2018.
https://doi.org/10.1007/978-3-319-96142-2_20

```
    Thread A        Thread B
---------------- -------------
+->assert(x==0);   step(1);
|                   ...
|                   ...
|                  step(m-1);
+------------------x = 1;
```

(a)

```
    Thread A          Thread B
--------------- -----------------
A1: x++;          B1: y--;
    barrier();        barrier();
A2: x--;          B2: x = 0;
A3: y++;          B3: y = 1;
```

(b)

Fig. 1. (a) An example illustrating random walk's weakness in probabilistic guarantee of error detection, where variable `x` is initially 0; (b) An example illustrating PCT's redundancy in exploring the partial order.

while still keeping the diversity of the interleavings explored [28]. The most straightforward sampling algorithm is *random walk*: at each step, randomly pick an *enabled* event to execute. Previous work showed that even such a sampling outperformed the exhaustive search at finding errors in real-world concurrent programs [24]. This can be explained by applying the *small-scope hypothesis* [12, Sect. 5.1.3] to the domain of concurrency error detection [17]: errors in real-world concurrent programs are non-adversarial and can often be triggered if a small number of events happen in the right order, which sampling has a good probability to achieve.

Random walk, however, has a unsurprisingly poor probabilistic guarantee of error detection. Consider the program in Fig. 1a. The assertion of thread `A` fails if, and only if, the statement "`x = 1`" of thread `B` is executed before this assertion. Without knowing which order (between the assertion and "`x = 1`") triggers this failure as a priori, we should sample both orders uniformly because the probabilistic guarantee of detecting this error is the *minimum* sampling probability of these two orders. Unfortunately, random walk may yield extremely non-uniform sampling probabilities for different orders when only a small number of events matter. In this example, to trigger the failure, the assertion of thread `A` has to be delayed (or not picked) by m times in random walk, making its probabilistic guarantee as low as $1/2^m$.

To sample different orders more uniformly, *Probabilistic Concurrency Testing* (PCT) [4] depends on a user-provided parameter d, the number of events to delay, to randomly pick d events within the execution, and inserts a preemption before each of the d events. Since the events are picked randomly by PCT, the corresponding interleaving space is sampled more uniformly, resulting in a much stronger probabilistic guarantee than random walk. Consider the program in Fig. 1a again. To trigger the failure, there is no event needed to be delayed, other than having the right thread (i.e. thread `B`) to run first. Thus, the probability trigger (or avoid) the failure is $1/2$, which is much higher than $1/2^m$.

However, PCT does not consider the partial order of events entailed by a concurrent program, such that the explored interleavings are still quite redundant. Consider the example in Fig. 1b. Both `A1` and `B1` are executed before the barrier and do not race with any statement. Statements `A2` and `B2` form a race, and so do statements `A3` and `B3`. Depending on how each race is resolved, the program events have total four different partial orders. However, without considering the effects of barriers, PCT will attempt to delay `A1` or `B1` in vain. Furthermore, without considering the race condition, PCT may first test an interleaving `A2` $\rightarrow$ `A3` $\rightarrow$ `B2` $\rightarrow$ `B3` (by delaying `A3` and `B2`), and then test a partial-order equivalent and thus completely redundant interleaving `A2` $\rightarrow$ `B2` $\rightarrow$ `A3` $\rightarrow$ `B3` (by delaying `A3` and `B3`). Such redundancies in PCT waste testing resources and weaken the probabilistic guarantee.

Towards addressing the above challenges, this paper makes three main contributions. First, we present a concurrency testing approach, named *partial order sampling* (POS), that samples the concurrent program execution based on the partial orders and provides strong probabilistic guarantees of error detection. In contrast to the sophisticated algorithms and heavy bookkeeping used in prior POR work, the core algorithm of POS is much more straightforward. In POS, each event is assigned with a random priority and, at each step, the event with the highest priority is executed. After each execution, all events that race with the executed event will be reassigned with a fresh random priority. Since each event has its own priority, POS (1) samples the orders of a group of dependent events uniformly and (2) uses one execution to sample independent event groups in parallel, both benefiting its probabilistic guarantee. The priority reassignment is also critical. Consider racing events e_1 and e_2, and an initial priority assignment that runs e_1 first. Without the priority reassignment, e_2 may very well be delayed again when a new racing event e_3 occurs because e_2's priority is more likely to be small (the reason that e_2 is delayed after e_1 at the first place). Such priority reassignments ensure that POS samples the two orders of e_2 and e_3 uniformly.

Secondly, the probabilistic guarantee of POS has been formally analyzed and shown to be exponentially stronger than random walk and PCT for general programs. The probability for POS to execute any partial order can be calculated by modeling the ordering constraints as a bipartite graph and computing the probability that these constraints can be satisfied by a random priority assignment. Although prior POR work typically have soundness proofs of the space reduction [1,8], those proofs depend on an exhaustive searching strategy and it is unclear how they can be adapted to randomized algorithms. Some randomized algorithms leverage POR to heuristically avoid redundant exploration, but no formal analysis of their probabilistic guarantee is given [22,28]. To the best of our knowledge, POS is the first work to sample partial orders with formal probabilistic guarantee of error detection.

Lastly, POS has been implemented and evaluated using both randomly generated programs and real-world concurrent software such as Firefox's JavaScript engine SpiderMonkey in SCTBench [24]. Our POS implementation supports shared-memory multithreaded programs using Pthreads. The evaluation results show that POS provided 134.1$\times$ stronger overall guarantees than random walk and PCT on randomly generated programs, and the error detection is 2.6$\times$ faster than random walk and PCT on SCTBench. POS managed to find the six most difficult bugs in SCTBench with the highest probability among all algorithms evaluated and performed the best among 20 of the total 32 non-trivial bugs in our evaluation.

Related Work. There is a rich literature of concurrency testing. Systematic testing [9,14,18,28] exhaustively enumerates all possible schedules of a program, which suffers from the state-space explosion problem. Partial order reduction techniques [1,2,8,10] alleviate this problem by avoiding exploring schedules that are redundant under partial order equivalence but rely on bookkeeping the massive exploration history to identify redundancy and it is unclear how they can be applied to the sampling methods.

PCT [4] explores schedules containing orderings of small sets of events and guarantees probabilistic coverage of finding bugs involving rare orders of a small number of events. PCT, however, does not take partial orders into account and becomes ineffective when dealing with a large number of ordering events. Also, the need of user-provided parameters diminishes the coverage guarantee, as the parameters are often provided imprecisely. Chistikov et al. [5] introduced hitting families to cover all admissible total orders of a set of events. However, this approach may cover redundant total orders that correspond to the same partial order. RAPOS [22] leverages the ideas from the partial order reduction, resembling our work in its goal, but does not provide a formal proof for its probabilistic guarantee. Our micro-benchmarks show that POS has a 5.0× overall advantage over RAPOS (see Sect. 6.1).

Coverage-driven concurrency testing [26,32] leverages relaxed coverage metrics to discover rarely explored interleavings. Directed testing [21,23] focuses on exploring specific types of interleavings, such as data races and atomicity violations, to reveal bugs. There is a large body of other work showing how to detect concurrency bugs using static analysis [19,25] or dynamic analysis [7,15,20]. But none of them can be effectively applied to real-world software systems, while still have formal probabilistic guarantees.

2 Running Example

Figure 2 shows the running example of this paper. In this example, we assume that memory accesses are sequentially consistent and all shared variables (e.g., `x`, `w`, etc.) are initialized to be 0. The program consists of two threads, i.e., `A` and `B`. Thread `B` will be blocked at `B4` by `wait(w)` until `w` > 0. Thread `A` will set `w` to be 1 at `A3` via `signal(w)` and unblock thread `B`. The assertion at `A4` will fail if, and only if, the program is executed in the following total order:

$$\texttt{B1} \rightarrow \texttt{A1} \rightarrow \texttt{B2} \rightarrow \texttt{B3} \rightarrow \texttt{A2} \rightarrow \texttt{A3} \rightarrow \texttt{B4} \rightarrow \texttt{B5} \rightarrow \texttt{B6} \rightarrow \texttt{A4}$$

To detect this bug, random walk has to make the correct choice at every step. Among all ten steps, three of them only have a single option: `A2` and `A3` must be executed first to enable `B4`, and `A4` is the only statement left at the last step. Thus, the probability of reaching the bug is $1/2^7 = 1/128$. As for PCT, we have

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

    Thread A             Thread B
-------------------- --------------------
                     local int a = b = 0;
A1: x++;             B1: x = 1;
A2: y++;             B2: a = x;
A3: signal(w);       B3: y = a;
A4: assert(z < 5);   B4: wait(w);
                     B5: b = y;
                     B6: z = a + b;
```

Fig. 2. The running example involving two threads.

to insert two preemption points just before statements B2 and A2 among ten statements, thus the probability for PCT is $1/10 \times 1/10 \times 1/2 = 1/200$, where this 1/2 comes from the requirement that thread B has to be executed first.

In POS, this bug can be detected with a substantial probability of 1/48, much higher than other approaches. Indeed, our formal guarantees ensure that any behavior of this program can be covered with a probability of at least 1/60.

3 Preliminary

Concurrent Machine Model. Our concurrent abstract machine models a *finite* set of processes and a set of shared objects. The machine state is denoted as s, which consists of the local state of each process and the state of shared objects. The abstract machine assumes the sequential consistency and allows the arbitrary interleaving among all processes. At each step, starting from s, any running process can be randomly selected to make a move to update the state to s' and generate an event e, denoted as $s \xrightarrow{e} s'$.

An event e is a tuple $e := (\texttt{pid}, \texttt{intr}, \texttt{obj}, \texttt{ind})$, where `pid` is the process ID, `intr` is the statement (or instruction) pointer, `obj` is the shared object accessed by this step (we assume each statement only access at most a single shared object), and `ind` indicates how many times this `intr` has been executed and is used to distinguish different runs of the same instruction. For example, the execution of the statement "`A2: y++`" in Fig. 2 will generate the event $(\texttt{A}, \texttt{A2}, \texttt{y}, \texttt{0})$. Such an event captures the information of the corresponding step and can be used to replay the execution. In other words, given the starting state s and the event e, the resulting state s' of a step "$\xrightarrow{e}$" is determined.

A trace t is a list of events generated by a sequence of program transitions (or steps) starting from the initial machine state (denoted as s_0). For example, the following program execution:

$$s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \cdots \xrightarrow{e_n} s_{n+1}$$

generates the trace $t := e_0 \bullet e_1 \bullet \cdots \bullet e_n$, where the symbol "$\bullet$" means"cons-ing" an event to the trace. Trace events can be accessed by index (e.g., $t[1] = e_1$).

A trace can be used to replay a sequence of executions. In other words, given the initial machine state s_0 and the trace t, the resulting state of running t (denoted as "$\texttt{State}(t)$") is determined.

We write $\texttt{En}(s) := \{e \mid \exists s', s \xrightarrow{e} s'\}$ as the set of events *enabled* (or allowed to be executed) at state s. Take the program in Fig. 2 as an example. Initially, both `A1` and `B1` can be executed, and the corresponding two events form the enabled set $\texttt{En}(s_0)$. The blocking wait at `B4`, however, can be enabled only after being signaled at `A3`. A state s is called a *terminating* state if, and only if, $\texttt{En}(s) = \emptyset$. We assume that any disabled event will eventually become enabled and every process must end with either a terminating state or an error state. This indicates that all traces are finite. For readability, we often abbreviate $\texttt{En}(\texttt{State}(t))$, i.e., the enabled event set after executing trace t, as $\texttt{En}(t)$.

Partial Order of Traces. Two events e_0 and e_1 are called *independent* events (denoted as $e_0 \perp e_1$) if, and only if, they neither belong to the same process nor access the same object:

$$e_0 \perp e_1 := \ (e_0.\texttt{pid} \neq e_1.\texttt{pid}) \ \wedge \ (e_0.\texttt{obj} \neq e_1.\texttt{obj})$$

The execution order of independent events does not affect the resulting state. If a trace t can be generated by swapping adjacent and independent events of another trace t', then these two traces t and t' are *partial order equivalent.* Intuitively, partial order equivalent traces are guaranteed to lead the program to the same state. The *partial order* of a trace is characterized by the orders between all *dependent* events plus their *transitive closure.* Given a trace t, its partial order relation "$\sqsubset_t$" is defined as the *minimal* relation over its events that satisfies:

$$\begin{aligned}
&(1)\ \forall i\ j,\ i < j\ \wedge\ t[i] \not\perp t[j] \implies t[i] \sqsubset_t t[j] \\
&(2)\ \forall i\ j\ k,\ t[i] \sqsubset_t t[j]\ \wedge\ t[j] \sqsubset_t t[k] \implies t[i] \sqsubset_t t[k]
\end{aligned}$$

Two traces with the same partial order relation and the same event set must be partial order equivalent.

Given an event order $\mathcal{E}$ and its order relation $\sqsubset_{\mathcal{E}}$, we say a trace t follows $\mathcal{E}$ and write "$t \simeq \mathcal{E}$" if, and only if,

$$\forall e_0\ e_1,\ e_0 \sqsubset_t e_1 \implies e_0 \sqsubset_{\mathcal{E}} e_1$$

We write "$t \models \mathcal{E}$" to denote that $\mathcal{E}$ is exactly the partial order of trace t:

$$t \models \mathcal{E} := \quad \forall e_0\ e_1,\ e_0 \sqsubset_t e_1 \iff e_0 \sqsubset_{\mathcal{E}} e_1$$

Probabilistic Error-Detection Guarantees. Each partial order of a concurrent program may lead to a different and potentially incorrect outcome. Therefore, any possible partial order has to be explored. The *minimum* probability of these explorations are called the probabilistic error-detection guarantee of a randomized scheduler.

Algorithm 1 presents a framework to formally reason about this guarantee. A sampling procedure Sample samples a terminating trace t of a program. It starts

Algorithm 1. Sample a trace using scheduler Sch and random variable R

```
1: procedure Sample(Sch, R)
2:     $t \leftarrow [\,]$
3:     while $\mathtt{En}(t) \neq \emptyset$ do
4:         $e \leftarrow \mathtt{Sch}(\mathtt{En}(t), \mathtt{R})$
5:         $t \leftarrow t \bullet e$
6:     end while
7:     return $t$
8: end procedure
```

with the empty trace and repeatedly invokes a randomized scheduler (denoted as Sch) to append an event to the trace until the program terminates. The randomized scheduler Sch selects an enabled event from $\mathtt{En}(t)$ and the randomness comes from the random variable parameter, i.e., R.

A naive scheduler can be purely random without any strategy. A sophisticated scheduler may utilize additional information, such as the properties of the current trace and the enabled event set.

Given the randomized scheduler Sch on R and any partial order $\mathcal{E}$ of a program, we write "$P(\text{Sample}(\mathtt{Sch}, \mathtt{R}) \models \mathcal{E})$" to denote the probability of covering $\mathcal{E}$, i.e., generating a trace whose partial order is exactly $\mathcal{E}$ using Algorithm 1. The *probabilistic error-detection guarantee* of the scheduler Sch on R is then defined as the minimum probability of covering the partial order $\mathcal{E}$ of any terminating trace of the program:

$$\min_{\mathcal{E}} P(\text{Sample}(\mathtt{Sch}, \mathtt{R}) \models \mathcal{E})$$

4 POS - Algorithm and Analysis

In this section, we first present BasicPOS, a priority-based scheduler and analyze its probability of covering a given partial order (see Sect. 4.1). Based on the analysis of BasicPOS, we then show that such a priority-based algorithm can be dramatically improved by introducing the *priority reassignment*, resulting in our POS algorithm (see Sect. 4.2). Finally, we present how to calculate the *probabilistic error-detection guarantee* of POS on general programs (see Sect. 4.3).

4.1 BasicPOS

In BasicPOS, each event is associated with a random and immutable priority, and, at each step, the enabled event with the highest priority will be picked to execute. We use Pri to denote the map from events to priorities and describe BasicPOS in Algorithm 2, which instantiates the random variable R in Algorithm 1 with Pri. The priority $\mathtt{Pri}(e)$ of every event e is independent with each other and follows the uniform distribution $\mathcal{U}(0, 1)$.

We now consider in what condition would BasicPOS sample a trace that follows a given partial order $\mathcal{E}$ of a program. It means that the generated trace t,

Algorithm 2. Sample a trace with BasicPOS under the priority map Pri

```
1: procedure $\text{Sample}_{\text{BasicPOS}}(\texttt{Pri})$        ▷ $\texttt{Pri} \sim \mathcal{U}(0,1)$
2:     $t \leftarrow [\,]$
3:     while $\texttt{En}(t) \neq \emptyset$ do
4:         $e^* \leftarrow \arg\max_{e \in \texttt{En}(t)} \texttt{Pri}(e)$
5:         $t \leftarrow t \bullet e^*$
6:     end while
7:     return $t$
8: end procedure
```

at the end of each loop iteration (line 5 in Algorithm 2), must satisfy the invariant "$t \simeq \mathcal{E}$". Thus, the event priorities have to be properly ordered such that, given a trace t satisfies "$t \simeq \mathcal{E}$", the enabled event e^* with the highest priority must satisfies "$t \bullet e^* \simeq \mathcal{E}$". In other words, given "$t \simeq \mathcal{E}$", for any $e \in \texttt{En}(t)$ and "$t \bullet e \not\simeq \mathcal{E}$", there must be some $e' \in \texttt{En}(t)$ satisfying "$t \bullet e' \simeq \mathcal{E}$" and a proper priority map where e' has a higher priority, i.e., $\texttt{Pri}(e') > \texttt{Pri}(e)$. Thus, e will not be selected as the event e^* at line 4 in Algorithm 2. The following Lemma 1 indicates that such an event e' always exists:

Lemma 1

$$\begin{aligned} &\forall t\ e,\ t \simeq \mathcal{E}\ \wedge\ e \in \texttt{En}(t)\ \wedge\ t \bullet e \not\simeq \mathcal{E} \\ &\implies \exists e',\ e' \in \texttt{En}(t)\ \wedge\ t \bullet e' \simeq \mathcal{E}\ \wedge\ e' \sqsubset_{\mathcal{E}} e \end{aligned}$$

Proof. We can prove it by contradiction. Since traces are finite, we assume that some traces are counterexamples to the lemma and t is the longest such trace. In other words, we have $t \simeq \mathcal{E}$ and there exists $e \in \texttt{En}(t) \wedge t \bullet e \not\simeq \mathcal{E}$ such that:

$$\forall e',\ e' \in \texttt{En}(t)\ \wedge\ t \bullet e' \simeq \mathcal{E}\ \implies \neg(e' \sqsubset_{\mathcal{E}} e) \tag{1}$$

Since $\mathcal{E}$ is the partial order of a terminating trace and the traces t has not terminated yet, we know that there must exist an event $e' \in \texttt{En}(t)$ such that $t \bullet e' \simeq \mathcal{E}$. Let $t' := t \bullet e'$, by (1), we have that $\neg(e' \sqsubset_{\mathcal{E}} e)$ and

$$\begin{aligned} & e \in \texttt{En}(t') \\ \wedge\ & t' \bullet e \not\simeq \mathcal{E} \\ \wedge\ & \forall e'',\ e'' \in \texttt{En}(t')\ \wedge\ t' \bullet e'' \simeq \mathcal{E}\ \implies \neg(e'' \sqsubset_{\mathcal{E}} e) \end{aligned}$$

First two statements are intuitive. The third one also holds, otherwise, $e' \sqsubset_{\mathcal{E}} e$ can be implied by the transitivity of partial orders using e''. Thus, t' is a counterexample that is longer than t, contradicting to our assumption. □

Thanks to Lemma 1, we then only need to construct a priority map such that this e' has a higher priority. Let "$e \bowtie_{\mathcal{E}} e' := \exists t,\ t \simeq \mathcal{E} \wedge \{e, e'\} \subseteq \texttt{En}(t)$" denote that e and e' can be *simultaneously enabled* under $\mathcal{E}$. We write

$$\texttt{PS}_{\mathcal{E}}(e) :=\ \{e' \mid e' \sqsubset_{\mathcal{E}} e\ \wedge\ e \bowtie_{\mathcal{E}} e'\}$$

as the set of events that can be simultaneously enabled with but have to be selected prior to e in order to follow $\mathcal{E}$. We have that any e' specified by Lemma 1 must belong to $\mathtt{PS}_{\mathcal{E}}(e)$. Let $V_{\mathcal{E}}$ be the event set ordered by $\mathcal{E}$. The priority map `Pri` can be constructed as below:

$$\bigwedge_{e \in V_{\mathcal{E}},\ e' \in \mathtt{PS}_{\mathcal{E}}(e)} \mathtt{Pri}(e) < \mathtt{Pri}(e') \qquad \text{(Cond-BasicPOS)}$$

The traces sampled by BasicPOS using this `Pri` will always follow $\mathcal{E}$.

Although (Cond-BasicPOS) is not the necessary condition to sample a trace following a desired partial order, from our observation, it gives a good estimation for the worst cases. This leads us to locate the major weakness of BasicPOS: the *constraint propagation* of priorities. An event e with a large $\mathtt{PS}_{\mathcal{E}}(e)$ set may have a relatively low priority since its priority has to be lower than all the events in $\mathtt{PS}_{\mathcal{E}}(e)$. Thus, for any simultaneously enabled event e' that has to be delayed after e, $\mathtt{Pri}(e')$ must be even smaller than $\mathtt{Pri}(e)$, which is unnecessarily hard to satisfy for a random $\mathtt{Pri}(e')$. Due to this constraints propagation, the probability that a priority map `Pri` satisfies (Cond-BasicPOS) can be as low as $1/|V_{\mathcal{E}}|!$.

Here, we explain how BasicPOS samples the following trace that triggers the bug described in Sect. 2:

$$\begin{aligned} t_{bug} := {} & (\mathtt{B},\mathtt{B1},\mathtt{x},\mathtt{0}) \bullet (\mathtt{A},\mathtt{A1},\mathtt{x},\mathtt{0}) \bullet (\mathtt{B},\mathtt{B2},\mathtt{x},\mathtt{0}) \bullet (\mathtt{B},\mathtt{B3},\mathtt{y},\mathtt{0}) \bullet (\mathtt{A},\mathtt{A2},\mathtt{y},\mathtt{0}) \\ & \bullet (\mathtt{A},\mathtt{A3},\mathtt{w},\mathtt{0}) \bullet (\mathtt{B},\mathtt{B4},\mathtt{w},\mathtt{0}) \bullet (\mathtt{B},\mathtt{B5},\mathtt{y},\mathtt{0}) \bullet (\mathtt{B},\mathtt{B6},\mathtt{z},\mathtt{0}) \bullet (\mathtt{A},\mathtt{A4},\mathtt{z},\mathtt{0}) \end{aligned}$$

To sample trace t_{bug}, according to (Cond-BasicPOS), the priority map has to satisfy the following constraints:

$$\begin{array}{llll} \mathtt{Pri}(t_{bug}[0] & = (\mathtt{B},\mathtt{B1},\mathtt{x},\mathtt{0})) & > \mathtt{Pri}(t_{bug}[1] & = (\mathtt{A},\mathtt{A1},\mathtt{x},\mathtt{0})) \\ \mathtt{Pri}(t_{bug}[1]) & & > \mathtt{Pri}(t_{bug}[2] & = (\mathtt{B},\mathtt{B2},\mathtt{x},\mathtt{0})) \\ \mathtt{Pri}(t_{bug}[2]) & & > \mathtt{Pri}(t_{bug}[4] & = (\mathtt{A},\mathtt{A2},\mathtt{y},\mathtt{0})) \\ \mathtt{Pri}(t_{bug}[3] & = (\mathtt{B},\mathtt{B3},\mathtt{y},\mathtt{0})) & > \mathtt{Pri}(t_{bug}[4]) & \\ \mathtt{Pri}(t_{bug}[6] & = (\mathtt{B},\mathtt{B4},\mathtt{w},\mathtt{0})) & > \mathtt{Pri}(t_{bug}[9] & = (\mathtt{A},\mathtt{A4},\mathtt{z},\mathtt{0})) \\ \mathtt{Pri}(t_{bug}[7] & = (\mathtt{B},\mathtt{B5},\mathtt{y},\mathtt{0})) & > \mathtt{Pri}(t_{bug}[9]) & \\ \mathtt{Pri}(t_{bug}[8] & = (\mathtt{B},\mathtt{B6},\mathtt{z},\mathtt{0})) & > \mathtt{Pri}(t_{bug}[9]) & \end{array}$$

Note that these are also the necessary constraints for BasicPOS to follow the partial order of t_{bug}. The probability that a random `Pri` satisfies the constraints is 1/120. The propagation of the constraints can be illustrated by the first three steps:

$$\mathtt{Pri}(t_{bug}[0]) > \mathtt{Pri}(t_{bug}[1]) > \mathtt{Pri}(t_{bug}[2]) > \mathtt{Pri}(t_{bug}[4])$$

that happens in the probability of 1/24. However, on the other hand, random walk can sample these three steps in the probability of 1/8.

4.2 POS

We will now show how to improve BasicPOS by eliminating the propagation of priority constraints. Consider the situation when an event e (delayed at some trace t) becomes eligible to schedule right after scheduling some e', i.e.,

$$t \simeq \mathcal{E} \ \wedge \ \{e, e'\} \subseteq \texttt{En}(t) \ \wedge \ t \bullet e \not\simeq \mathcal{E} \ \wedge \ t \bullet e' \bullet e \simeq \mathcal{E}$$

If we reset the priority of e right after scheduling e', all the constraints causing the delay of e will not be propagated to the event e'' such that $e \in \texttt{PS}_{\mathcal{E}}(e'')$. However, there is no way for us to know which e should be reset after e' during the sampling, since $\mathcal{E}$ is unknown and not provided. Notice that

$$t \simeq \mathcal{E} \ \wedge \ \{e, e'\} \subseteq \texttt{En}(t) \ \wedge \ t \bullet e \not\simeq \mathcal{E} \ \wedge \ t \bullet e' \bullet e \simeq \mathcal{E} \implies e.\texttt{obj} = e'.\texttt{obj}$$

If we reset the priority of all the events that access the same object with e', the propagation of priority constraints will also be eliminated.

To analyze how POS works to follow $\mathcal{E}$ under the reassignment scheme, we have to model how many priorities need to be reset at each step. Note that blindly reassigning priorities of all delayed events at each step would be sub-optimal, which degenerates the algorithm to random walk. To give a formal and more precise analysis, we introduce the object index functions for trace t and partial order $\mathcal{E}$:

$$\begin{aligned}\texttt{I}(t, e) &:= \ |\{e' \mid e' \in t \ \wedge \ e.\texttt{obj} = e'.\texttt{obj}\}| \\ \texttt{I}_{\mathcal{E}}(e) &:= \ |\{e' \mid e' \sqsubset_{\mathcal{E}} e \ \wedge \ e.\texttt{obj} = e'.\texttt{obj}\}|\end{aligned}$$

Intuitively, when $e \in \texttt{En}(t)$, scheduling e on t will operate $e.\texttt{obj}$ after $\texttt{I}(t, e)$ previous events. A trace t follows $\mathcal{E}$ if every step (indicated by $t[i]$) operates the object $t[i].\texttt{obj}$ after $\texttt{I}_{\mathcal{E}}(t[i])$ previous events in the trace.

We then index (or version) the priority of event e using the index function as $\texttt{Pri}(e, \texttt{I}(t, e))$ and introduce POS shown in Algorithm 3. By proving that

$$\forall e', \ \texttt{I}(t, e) \leq \texttt{I}(t \bullet e', e) \ \wedge \ (\texttt{I}(t, e) = \texttt{I}(t \bullet e', e) \iff e.\texttt{obj} \neq e'.\texttt{obj})$$

we have that scheduling an event e will *increase* the priority version of all the events accessing $e.\texttt{obj}$, resulting in the priority reassignment.

We can then prove that the following statements hold:

$$\begin{aligned}&\forall t \ e, \ t \simeq \mathcal{E} \wedge e \in \texttt{En}(t) \implies (t \bullet e \simeq \mathcal{E} \iff \texttt{I}(t, e) = \texttt{I}_{\mathcal{E}}(e)) \\ &\forall t \ e, \ t \simeq \mathcal{E} \wedge e \in \texttt{En}(t) \wedge t \bullet e \not\simeq \mathcal{E} \implies \texttt{I}(t, e) < \texttt{I}_{\mathcal{E}}(e)\end{aligned}$$

To ensure that the selection of e^* on trace t follows $\mathcal{E}$ at the line 4 of Algorithm 3, any e satisfying $\texttt{I}(t, e) < \texttt{I}_{\mathcal{E}}(e)$ has to have a smaller priority than some e' satisfying $\texttt{I}(t, e') = \texttt{I}_{\mathcal{E}}(e)$ and such e' must exist by Lemma 1. In this way, the priority constraints for POS to sample $\mathcal{E}$ are as below:

$$\bigwedge \texttt{Pri}(e, i) < \texttt{Pri}(e', \texttt{I}_{\mathcal{E}}(e')) \text{ for some } i < \texttt{I}_{\mathcal{E}}(e)$$

which is bipartite and the propagation of priority constraints is eliminated. The effectiveness of POS is guaranteed by Theorem 1.

Algorithm 3. Sample a trace with POS under versioned priority map `Pri`

```
1: procedure Sample_POS(Pri)                                   ▷ $\texttt{Pri} \sim \mathcal{U}(0,1)$
2:     $t \leftarrow [\,]$
3:     while $\texttt{En}(t) \neq \emptyset$ do
4:         $e^* \leftarrow \arg\max_{e \in \texttt{En}(t)} \texttt{Pri}(e, \texttt{I}(t,e))$
5:         $t \leftarrow t.e^*$
6:     end while
7:     return $t$
8: end procedure
```

Theorem 1. *Given any partial order $\mathcal{E}$ of a program with $\mathcal{P} > 1$ processes. Let*

$$D_{\mathcal{E}} := |\,\{(e, e') \mid e \sqsubset_{\mathcal{E}} e' \ \wedge\ e \not\preceq e' \ \wedge\ e \bowtie_{\mathcal{E}} e'\}\,|$$

be the number of races in $\mathcal{E}$, we have that

1. *$D_{\mathcal{E}} \leq |V_{\mathcal{E}}| \times (\mathcal{P} - 1)$, and*
2. *POS has at least the following probability to sample a trace $t \simeq \mathcal{E}$:*

$$\left(\frac{1}{\mathcal{P}}\right)^{|V_{\mathcal{E}}|} R^U$$

where $R = \mathcal{P} \times |V_{\mathcal{E}}|/(|V_{\mathcal{E}}| + D_{\mathcal{E}}) \geq 1$ and $U = (|V_{\mathcal{E}}| - \lceil D_{\mathcal{E}}/(\mathcal{P} - 1)\rceil)/2 \geq 0$

Please refer to the technical report [33] for the detailed proof and the construction of priority constraints.

Here, we show how POS improves BasicPOS over the example in Sect. 2. The priority constraints for POS to sample the partial order of t_{bug} are as below:

$$\begin{aligned}
\texttt{Pri}(t_{bug}[0]\,,0) &> \texttt{Pri}(t_{bug}[1]\,,0)\\
\texttt{Pri}(t_{bug}[1]\,,1) &> \texttt{Pri}(t_{bug}[2]\,,1)\\
\texttt{Pri}(t_{bug}[2]\,,2) &> \texttt{Pri}(t_{bug}[4]\,,0)\\
\texttt{Pri}(t_{bug}[3]\,,0) &> \texttt{Pri}(t_{bug}[4]\,,0)\\
\texttt{Pri}(t_{bug}[6]\,,1) &> \texttt{Pri}(t_{bug}[9]\,,0)\\
\texttt{Pri}(t_{bug}[7]\,,2) &> \texttt{Pri}(t_{bug}[9]\,,0)\\
\texttt{Pri}(t_{bug}[8]\,,0) &> \texttt{Pri}(t_{bug}[9]\,,0)
\end{aligned}$$

Since each $\texttt{Pri}(e, i)$ is independently random following $\mathcal{U}(0, 1)$, the probability of `Pri` satisfying the constraints is $1/2 \times 1/2 \times 1/3 \times 1/4 = 1/48$.

4.3 Probability Guarantee of POS on General Programs

We now analyze how POS performs on general programs compared to random walk and PCT. Consider a program with $\mathcal{P}$ processes and $\mathcal{N}$ total events. It is generally common for a program have substantial non-racing events, for example, accessing shared variables protected by locks, semaphores, and condition variables, etc. We assume that there exists a ratio $0 \leq \alpha \leq 1$ such that in any partial order there are at least $\alpha\mathcal{N}$ non-racing events.

Under this assumption, for random walk, we can construct an adversary program with the worst case probability as $1/\mathcal{P}^{\mathcal{N}}$ for almost any α [33]. For PCT, since only the order of the $(1-\alpha)\mathcal{N}$ events may affect the partial order, the number of preemptions needed for a partial order in the worst case becomes $(1-\alpha)\mathcal{N}$, and thus the worst case probability bound is $1/\mathcal{N}^{(1-\alpha)\mathcal{N}}$. For POS, the number of races $D_{\mathcal{E}}$ is reduced to $(1-\alpha)\mathcal{N} \times (\mathcal{P}-1)$ in the worst case, Theorem 1 guarantees the probability lower bound as

$$\frac{1}{\mathcal{P}^{\mathcal{N}}}\left(\frac{1}{1-(1-1/P)\alpha}\right)^{\alpha\mathcal{N}/2}$$

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

5 Implementation

The algorithm of POS requires a pre-determined priority map, while the implementation could decide the event priority on demand when new events appear. The implementation of POS is shown in Algorithm 4, where lines 14–18 are for the priority reassignment. Variable s represents the current program state with the following interfaces:

– s.Enabled() returns the current set of enabled events.
– s.Execute(e) returns the resulting state after executing e in the state of s.
– s.IsRacing(e, e') returns if there is a race between e and e'.

In the algorithm, if a race is detected during the scheduling, the priority of the delayed event in the race will be removed and then be reassigned at lines 6–9.

Relaxation for Read-Only Events. The abstract interface s.IsRacing($\ldots$) allows us to relax our model for read-only events. When both e and e' are read-only events, s.IsRacing(e, e') returns false even if they are accessing the same object. Our evaluations show that this relaxation improves the execution time of POS.

Fairness Workaround. POS is probabilistically fair. For an enabled event e with priority $p > 0$, the cumulative probability for e to delay by $k \to \infty$ steps without racing is at most $(1-p^{\mathcal{P}})^k \to 0$. However, it is possible that POS delays events for prolonged time, slowing down the test. To alleviate this, the current implementation resets all event priorities for every 10^3 voluntary context switch events, e.g., `sched_yield()` calls. This is only useful for speeding up few benchmark programs that have busy loops (`sched_yield()` calls were added by SCTBench creators) and has minimal impact on the probability of hitting bugs.

Algorithm 4. Testing a program with POS

```
1: procedure POS($s$)                                   ▷ $s$: the initial state of the program
2:     $pri \leftarrow [\epsilon \mapsto -\infty]$ ▷ Initially, no priority is assigned except the special symbol $\epsilon$
3:     while $s$.Enabled() $\neq \emptyset$ do
4:         $e^* \leftarrow \epsilon$                                ▷ Assume $\epsilon \notin s$.Enabled()
5:         for each $e \in s$.Enabled() do
6:             if $e \notin pri$ then
7:                 $newPriority \leftarrow \mathcal{U}(0,1)$
8:                 $pri \leftarrow pri[e \mapsto newPriority]$
9:             end if
10:            if $pri(e^*) < pri(e)$ then
11:                $e^* \leftarrow e$
12:            end if
13:        end for
14:        for each $e \in s$.Enabled() do                  ▷ Update priorities
15:            if $e \neq e^* \wedge s$.IsRacing($e, e^*$) then
16:                $pri \leftarrow pri \setminus \{e\}$     ▷ The priority will be reassigned in the next step
17:            end if
18:        end for
19:        $s \leftarrow s$.Execute($e^*$)
20:    end while
21:    return $s$
22: end procedure
```

6 Evaluation

To understand the performance of POS and compare with other sampling methods, we conducted experiments on both micro benchmarks (automatically generated) and macro benchmarks (including real-world programs).

6.1 Micro Benchmark

We generated programs with a small number of static events as the micro benchmarks. We assumed multi-threaded programs with t threads and each thread executes m events accessing o objects. To make the program space tractable, we chose $t = m = o = 4$, resulting 16 total events. To simulate different object access patterns in real programs, we chose to randomly distribute events accessing different objects with the following configurations:

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

The results are shown in Table 1. The benchmark columns show the characteristics of each generated program, including (1) the configuration used for generating the program; (2) the number of distinct partial orders in the program; (3) the maximum number of preemptions needed for covering all partial orders; and (4) the maximum number of races in any partial order. We measured the

Table 1. Coverage on the micro benchmark programs. Columns under "benchmark" are program characteristics explained in Sect. 6.1. "0(x)" represents incomplete coverage.

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

Table 2. Coverage on the micro benchmark programs - 50% read

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

coverage of each sampling method on each program by the minimum hit ratio on any partial order of the program. On every program, we ran each sampling methods for 5×10^7 times (except for random walk, for which we calculated the exact probabilities). If a program was not fully covered by an algorithm within the sample limit, the coverage is denoted as "0(x)", where x is the number of covered partial orders. We let PCT sample the exact number of the preemptions needed for each case. We tweaked PCT to improve its coverage by adding a dummy event at the beginning of each thread, as otherwise PCT cannot preempt the actual first event of each thread. The results show that POS performed the best among all algorithms. For each algorithm, we calculated the overall performance as the geometric mean of the coverage.[1] POS overall performed $\sim$7.0$\times$ better compared to other algorithms ($\sim$134.1$\times$ excluding RAPOS and BasicPOS).

To understand our relaxation of read-only events, we generated another set of programs with the same configurations, but with half of the events read-only. The results are shown in Table 2, where the relaxed algorithm is denoted as POS*. Overall, POS* performed roughly $\sim$1.4$\times$ as good as POS and $\sim$5.0$\times$ better compared to other algorithms ($\sim$226.4$\times$ excluding RAPOS and BasicPOS).

6.2 Macro Benchmark

We used SCTBench [24], a collection of concurrency bugs on multi-threaded programs, to evaluate POS on practical programs. SCTBench collected 49 concurrency bugs from previous parallel workloads [3,27] and concurrency testing/verification work [4,6,18,21,31]. SCTBench comes with a concurrency testing tool, Maple [32], which intercepts `pthread` primitives and shared memory accesses, as well as controls their interleaving. When a bug is triggered, it will be caught by Maple and reported back. We implemented POS with the relaxation of read-only events in Maple. Each sampling method was evaluated in SCTBench by the ratio of tries and hits of the bug in each case. For each case, we ran each sampling method on it until the number of tries reaches 10^4. We recorded the bug hit count h and the total runs count t, and calculated the ratio as h/t.

Two cases in SCTBench are not adopted: `parsec-2.0-streamcluster2` and `radbench-bug1`. Because neither of the algorithms can hit their bugs once, which conflicts with previous results. We strengthened the case `safestack-bug1` by internally repeating the case for 10^4 times (and shrunk the run limit to 500). This amortizes the per-run overhead of Maple, which could take up to a few seconds. We modified PCT to reset for every internal loop. We evaluated variants of PCT algorithms of PCT-d, representing PCT with $d-1$ preemption points, to reduce the disadvantage of a sub-optimal d. The results are shown in Table 3. We ignore cases in which all algorithms can hit the bugs with more than half of their tries. The cases are sorted based on the minimum hit ratio across algorithms. The performance of each algorithm is aggregated by calculating the geometric mean of hit ratios[2] on every case. The best hit ratio for each case is marked as blue.

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

- Overall, POS performed the best in hitting bugs in SCTBench. The geometric mean of POS is $\sim$2.6$\times$ better than PCT and $\sim$4.7$\times$ better than random walk. Because the buggy interleavings in each case are not necessarily the most

[1] For each case that an algorithm does not have the full coverage, we conservatively account the coverage as $\frac{1}{5\times10^7}$ into the geometric mean.

[2] For each case that an algorithm cannot hit once within the limit, we conservatively account the hit ratio as $1/t$ in the calculation of the geometric mean.

difficult ones to sample, POS may not perform overwhelmingly better than others, as in micro benchmarks.

- Among all 32 cases shown in the table, POS performed the best among all algorithms in 20 cases, while PCT variants were the best in 10 cases and random walk was the best in three cases.
- POS is able to hit all bugs in SCTBench, while all PCT variants missed one case within the limit (and one case with hit ratio of 0.0002), and random walk missed three cases (and one case with hit ratio of 0.0003).

Table 3. Bug hit ratios on macro benchmark programs

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

7 Conclusion

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

Acknowledgements. We thank the anonymous reviewers for their helpful feedbacks that greatly improved this paper. We thank Madan Musuvathi for insightful discussions. This research was supported in part by NSF CNS-1564055, ONR N00014-16-1-2263, and ONR N00014-17-1-2788 grants.

References

1. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 373–384. ACM, San Diego (2014)
2. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 526–543. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_26
3. Bienia, C.: Benchmarking Modern Multiprocessors. Princeton University, Princeton (2011)
4. Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. In: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV, pp. 167–178. ACM, Pittsburgh (2010)
5. Chistikov, D., Majumdar, R., Niksic, F.: Hitting families of schedules for asynchronous programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 157–176. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_9
6. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 331–340. ACM, Waikiki (2011)
7. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 121–133. ACM, Dublin (2009)
8. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, pp. 110–121. ACM, Long Beach (2005)

9. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, pp. 174–186. ACM, Paris (1997)
10. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer, New York (1996). https://doi.org/10.1007/3-540-60761-7
11. Guo, H., Wu, M., Zhou, L., Hu, G., Yang, J., Zhang, L.: Practical software model checking via dynamic interface reduction. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 2011, pp. 265–278. ACM, Cascais (2011)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
13. Jackson, J.: Nasdaq's Facebook Glitch Came From Race Conditions. http://www.cio.com/article/706796/
14. Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J.F., Gunawi, H.S.: SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI 2014, pp. 399–414. USENIX Association, Broomfield (2014)
15. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, pp. 37–48. ACM, San Jose (2006)
16. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: a pragmatic approach to model checking real code. In: Proceedings of the 5th Symposium on Operating Systems Design and implementation, OSDI 2002, pp. 75–88. USENIX Association, Boston (2002)
17. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 446–455. ACM, San Diego (2007)
18. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008, pp. 267–280. USENIX Association, San Diego (2008)
19. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2006, pp. 308–319. ACM, Ottawa (2006)
20. O'Callahan, R., Choi, J.-D.: Hybrid dynamic data race detection. In: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2003, pp. 167–178. ACM, San Diego (2003)
21. Park, S., Lu, S., Zhou, Y.: CTrigger: exposing atomicity violation bugs from their hiding places. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, pp. 25–36. ACM, Washington, D.C. (2009)
22. Sen, K.: Effective random testing of concurrent programs. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE 2007, pp. 323–332. ACM, Atlanta (2007)
23. Sen, K.: Race directed random testing of concurrent programs. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 11–21. ACM, Tucson (2008)

24. Thomson, P., Donaldson, A.F., Betts, A.: Concurrency testing using controlled schedulers: an empirical study. ACM Trans. Parallel Comput. **2**(4), 23:1–23:37 (2016)
25. Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE 2007, pp. 205–214. ACM, Dubrovnik (2007)
26. Wang, C., Said, M., Gupta, A.: Coverage guided systematic concurrency testing. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 221–230. ACM, Waikiki (2011)
27. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA 1995, pp. 24–36. ACM, S. Margherita Ligure (1995)
28. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: transparent model checking of unmodified distributed systems. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, pp. 213–228. USENIX Association, Boston (2009)
29. Yang, J., Sar, C., Engler, D.: EXPLODE: a lightweight, general system for finding serious storage system errors. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7, OSDI 2006, p. 10. USENIX Association, Seattle (2006)
30. Yang, J., Twohey, P., Engler, D., Musuvathi, M.: Using model checking to find serious file system errors. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI 2004, p. 19. USENIX Association, San Francisco (2004)
31. Yang, Y., Chen, X., Gopalakrishnan, G.: Inspect: a runtime model checker for multithreaded C programs. Technical report UUCS-08-004, University of Utah, Salt Lake City, UT, USA (2008)
32. Yu, J., Narayanasamy, S., Pereira, C., Pokam, G.: Maple: a coverage-driven testing tool for multithreaded programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 485–502. ACM, Tucson (2012)
33. Yuan, X., Yang, J., Gu, R.: Partial order aware concurrency sampling (extended version). Technical report, Columbia University, New York, NY, USA (2018)