Testing Multithreaded Programs via Thread Speed Control*

Dongjie Chen
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
midwinter1993@gmail.com

Yanyan Jiang
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
jyy@nju.edu.cn

Chang Xu
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
changxu@nju.edu.cn

Xiaoxing Ma
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
xxm@nju.edu.cn

Jian Lu
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
lj@nju.edu.cn

ABSTRACT

A multithreaded program's interleaving space is discrete and astronomically large, making effectively sampling thread schedules for manifesting concurrency bugs a challenging task. Observing that concurrency bugs can be manifested by adjusting thread relative speeds, this paper presents the new concept of *speed space* in which each vector denotes a family of thread schedules. A multithreaded program's speed space is approximately continuous, easy-to-sample, and preserves certain categories of concurrency bugs. We discuss the design, implementation, and evaluation of our speed-controlled scheduler for exploring adversarial/abnormal schedules. The experimental results confirm that our technique is effective in sampling diverse schedules. Our implementation also found previously unknown concurrency bugs in real-world multithreaded programs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Concurrency, testing, interleaving space sampling, race detection

ACM Reference Format:
Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. 2018. Testing Multithreaded Programs via Thread Speed Control. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236077

*Yanyan Jiang (jyy@nju.edu.cn) and Chang Xu (changxu@nju.edu.cn) are the corresponding authors.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236077

1 INTRODUCTION

Multithreaded programs are prevalent but also error prone. Concurrency bugs have caused serious real-world consequences, e.g., the Northeast blackout [32] and mismatched NASDAQ Facebook share prices [30].

Unlike sequential bugs, the manifestation of concurrency bugs depends not only on inputs, but also on *thread schedules* in an interleaving space, which is exponential of a program's execution trace length. This makes finding concurrency bugs difficult–they usually can only be triggered by specific schedules with certain event orderings.

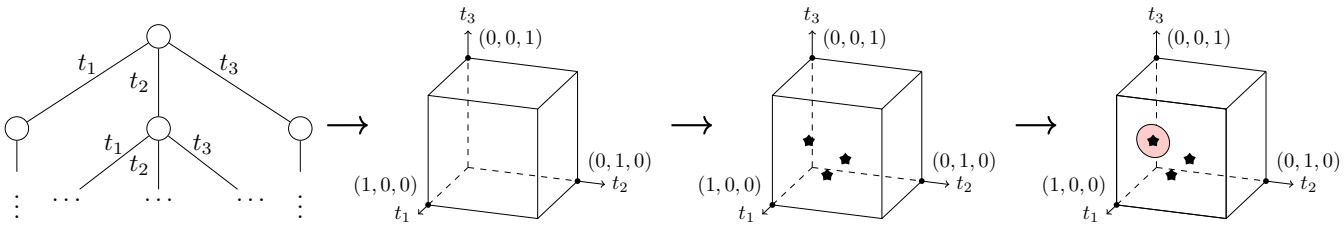
An exhaustive exploration of the interleaving space (e.g., by model checking [14]) to find buggy schedules is usually infeasible for large real-world programs. Rather, one often *samples* schedules randomly [5], by heuristics [34], or in a bounded interleaving space [26]. However, these approaches waste resources in exercising many schedules with the same fixed event orderings, leaving concurrency bugs undetected within limited testing budgets.

In this paper, we present an alternative representation of the interleaving space (named *speed space*), which is approximately continuous, easy-to-sample, and preserves certain categories of concurrency bugs. The speed space helps us exercise adversarial/abnormal schedules more easily and develop effective testing techniques.

Our testing technique based on a speed-controlled scheduler is motivated by the following key observations:

(1) Events usually happen in a particular order under a native scheduler because threads run with almost the same execution speeds [28], which by its nature leaves many concurrency bugs not manifested.
(2) Events leading to many concurrency bugs in real systems are coarsely interleaved [18]. Therefore, concurrency bugs can be manifested by a coarse-grained scheduler.
(3) Interleaving space sampling and predictive analysis [13, 19] complement each other. They can be combined to yield an effective testing technique for multithreaded programs.

The first observation suggests that to effectively sample schedules in an interleaving space, event orderings enforced by a native scheduler should be intentionally reversed. This is achieved by *projecting* the discrete interleaving space (with loss) to an approximately continuous *"speed" space* (in Figure 1b) in which each vector



(a) Interleaving space **(b) Speed space** **(c) Sample schedules systematically** **(d) Extend with trace-based analysis**

Figure 1: Testing multithreaded programs via thread speed control: An overview.

denotes a set of schedules whose threads run in certain relative speeds. This projection not only reduces the space but also makes sampling adversarial/abnormal schedules easier (in Figure 1c).

The second observation, also known as the coarse interleaving hypothesis [18], implies that *the projected speed space can be sampled in a coarse-grained level* for manifesting concurrency bugs. We found that the speed space can be further reduced by only considering function-call schedules instead of read/write event schedules, while still being capable of manifesting concurrency bugs.

The final observation indicates that the bug-finding capability of interleaving space sampling can be further *enhanced by applying predictive analyses* [19] (in Figure 1d). A predictive analysis explores a larger set of schedules (usually defined by a causal-model [17]) in the interleaving space given a seed schedule. Feeding such an analysis with diverse seeds (sampled schedules in the interleaving space) helped us reveal long-time hidden concurrency bugs in real-world programs.

In summary, we test a multithreaded program by sampling in its reduced speed space and applying a predictive analysis (in Figure 1). These ideas are implemented as a prototype tool–"Schnauzer" built upon LLVM [2]. The experiments show that our sampling technique manifested schedules with more happens-before races in most benchmarks, varying from 9% to 172%. Schnauzer also found previously unknown concurrency bugs in Cherokee Web Server [1] and Transmission Client [3].

The contributions of this paper are listed as follows:

(1) We introduced the *speed space*, an alternative reduced representation of a multithreaded program's interleaving space. It is easier to reverse event orderings and sample schedules in this approximately continuous space.
(2) We proposed an algorithm to systematically sample the speed space for diverse thread schedules.
(3) We discovered that speed space sampling can be made more practical by leveraging the coarse interleaving hypothesis and predictive analyses.
(4) We implemented the ideas as a prototype tool and evaluated it, with previously unknown concurrency bugs found in real-world open-source multithreaded programs.

This paper is laid out as follows. Section 2 introduces the background and motivation. Section 3 defines the speed space and elaborates on how to sample schedules systematically in the speed space. Section 4 gives an overview of our prototype tool and implementation, followed by the evaluation in Section 5. Section 6 describes related work and Section 7 concludes this paper.

2 BACKGROUND AND MOTIVATION

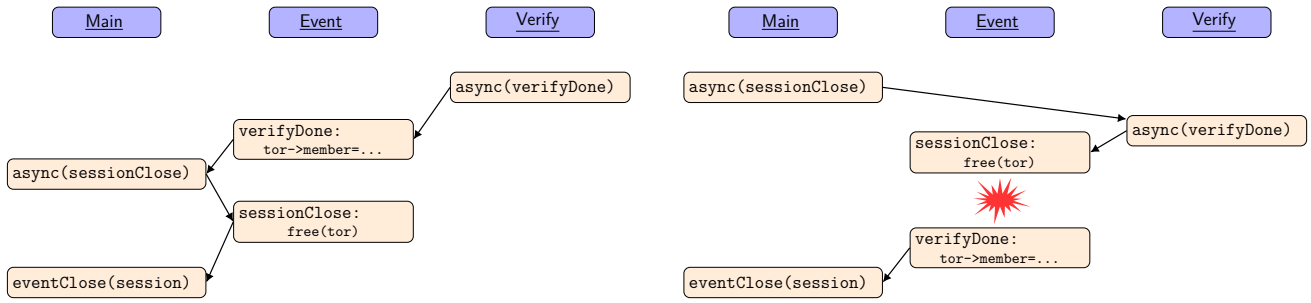
2.1 Testing Multithreaded Programs

Concurrency bugs are notoriously hard to detect because they may only be triggered by specific schedules, while the interleaving space of a multithreaded program contains an astronomical number of schedules. Ideally, a multithreaded program should be exhaustively tested, e.g., using model checking techniques [12] that enumerate all possible program states (and schedules) to find buggy schedules. Windowing techniques [17, 36] and bounding techniques [11, 26] are widely used to check a portion of schedules for making model checking more practically applicable. The former attempts to divide the execution into a sequence of fixed-size windows and perform analysis on each window separately; whereas the latter bounds the state space to explore. Random testing techniques are also proposed to *sample* a small portion of schedules in the interleaving space. For example, PCT (Probabilistic Concurrency Testing) [5] can detect concurrency bugs of *bug depth d* (i.e., the minimum number of specific event orderings sufficient to find the bug) with probability at least $1/nk^{d-1}$ for a multithreaded program consisting of n threads and k execution steps.

2.2 Motivating Example

Consider a concurrency bug in the Transmission Client [3][1] as illustrated in Figure 2[2]. The operating system's native scheduler almost definitely schedules the request sent by the `verify` thread before the `main` thread because `verifyDone` is usually invoked at the beginning of the execution while `sessionClose` is called at the end, forming the event ordering in Figure 2a. In some occasions (Figure 2b) which were previously unknown to the developers, `verifyDone` can be scheduled earlier than `sessionClose`, yielding a use-after-free on `tor`. To manifest this bug, three event orderings are required: the `main` thread must be scheduled to send requests

[1] https://github.com/transmission/transmission/issues/423. The bug was *previously unknown*. It was first found by the technique described in this paper.
[2] The visual convention is that a blue box denotes the thread name and a yellow box denotes an event. Events in a thread chronologically happen in the top-down order. A black arrow between events e_1 and e_2 denotes that e_1 happened before e_2.



(a) Typical event orderings under a native scheduler

(b) The buggy schedule with a use-after-free on `tor`

Figure 2: The motivating example of a concurrency bug in Transmission Client

earlier than the `verify` thread, and the use-after-free must happen before `eventClose`, otherwise the program will be terminated.

Unfortunately, state-of-the-art testing techniques may be ineffective in detecting such complex concurrency bugs. For PCT [5], the fact that the bug depth $d = 3$ implies that one expects to find the bug within seemingly promising $O(nk^2)$ runs. However, since k is large for an execution trace (in our experiment, k is approximately two million), it may need trillions of runs to strike the delicate priority-changing points. Model checking [12] also becomes intractable for these practical programs with long traces. Limiting the checked trace in a small window [17, 36] solves the tractability issue but still misses the bug because `verifyDone` (near the beginning of an execution) and `sessionClose` (near the end of an execution) are distant in the trace under the native scheduler. Even worse, since there is no data race in the schedule (both the free and use operation happen in the same thread), race-directed techniques [34] cannot capture the bug.

2.3 The Need for an Alternative Representation of the Interleaving Space

Finding these buggy schedules in the huge interleaving space is a challenge. The most straightforward approach to exploring the interleaving space is to traverse its state transition tree in which a node u denotes a program state and an edge (u, v) indicates that v is the state obtained by a thread executing one step from u, as shown Figure 1a. The state tree is the basis of many (stateful or stateless) model checking techniques [12, 15].

On the other hand, both PCT [5] and the delay-bound scheduler [11] ingeniously transform the state tree (in which each node may have several children) into a *binary tree*. For most of the execution steps, these two techniques simply let the current running thread proceed, and perform context switches (or preemptions) at certain program points. This transformation helps sample schedules in the interleaving space *more easily* (a path/trace can be encoded as a bit string) and *more effectively*. Both techniques take samples in a small subspace in which the number of ones in the bit string is constrained. Such a small subspace denotes schedules consisting of just a few context switches, and leverages the fact that many practical concurrency bugs can be manifested in such a condition.

Realizing the power of interleaving space transformation, one also raises a question: can we transform the interleaving space to an *approximately continuous* space that is even easier to effectively sample? This paper gives a positive answer by trying to *control each thread's "execution speed"*.

In the motivating example, if we slow down the `verify` thread, the `main` thread will send `sessionClose` request later, and then, the `event` thread will reclaim memory in `sessionClose`. Furthermore, if one lets the `main` thread run slower than the `event` thread, it stands a good chance that the `main` thread will exit after the `event` thread. At the moment the `verifyDone` function is invoked, the use-after-free bug is triggered.

This paper presents an approach to manifesting concurrency bugs by systematically sampling a *hypercube* (Figure 1b) containing threads' relative speed vectors.

3 DEFINING AND SAMPLING THE SPEED SPACE

3.1 Notations and Definitions

A multithreaded program P consists of a set of concurrent *threads* $T = \{t_1, \ldots, t_n\}$. Threads are executed by a *scheduler* $\mathcal{S}$ by repeatedly choosing a thread to perform a series of thread-local computations and an operation on a shared object (a shared memory variable or a lock variable). Each operation on a shared object is associated with an *event* e of the following types:

- $\mathsf{Read}(t, x)/\mathsf{Write}(t, x)$ for thread t accessing the shared memory variable x.
- $\mathsf{Lock}(t, \ell)/\mathsf{Unlock}(t, \ell)$ for thread t performing a synchronization operation on the lock variable ℓ;
- $\mathsf{Fork}(t, u)/\mathsf{Join}(t, u)$ for thread t forking/joining a thread u;
- Nop for doing nothing when thread t is blocked while being chosen to execute[3].

We use $e.op \in \{\mathsf{Read}, \mathsf{Write}, \mathsf{Lock}, \mathsf{Unlock}, \mathsf{Fork}, \mathsf{Join}\}$, $e.t \in T$, and $e.x$ to denote the operation type, the thread, and the variable associated with an event e. Chronologically concatenating all events in a program execution yields an *execution trace* τ, a sequence of events $\tau = \langle e_1, \ldots e_m \rangle$. We denote events in τ that are performed by thread t as $\tau\restriction_t$, and use $\tau_{i:j} = \langle e_i, e_{i+1}, \ldots, e_j \rangle$ to denote a substring of τ. The *schedule* of a trace τ, $\mathsf{sched}(\tau)$, is defined as the sequence of thread identifiers in τ: $\mathsf{sched}(\tau) = \langle e_1.t, e_2.t, \ldots, e_m.t \rangle$.

[3] We introduce Nop for an easier description of the semantics in a speed-controlled scheduler.

The interleaving space Σ of a multithreaded program is the set of all execution traces' schedules. For a program with n threads and m events to execute, there are at most m^n execution traces, i.e., $|\Sigma| = O(m^n)$.

We observed that concurrency bugs are triggered by particular event orderings, however, one usually has little knowledge of which events may relate to concurrency bugs. Furthermore, given a prefix of an execution trace τ, it is intractable to predict the subsequent events in τ. These observations make sampling of Σ challenging: even if we have collected a set of traces in testing, the criterion for guiding the selection of the *next* schedule is unclear (and therefore existing techniques are mostly supported by heuristics [5, 8, 11, 34]). We design our alternative representation of Σ such that schedules are easy to generate while buggy schedules are still preserved.

3.2 The Speed Space

The idea that concurrency bugs are triggered by specific event orderings inspired us to control the *relative speed* between threads to manifest them. For a multithreaded program P of n threads, a *speed vector* $\gamma = [\gamma_1, \ldots, \gamma_n]$ denotes that thread t_i is assigned with a *speed* of $\gamma_i \in (0, 1]$.

A speed vector γ represents schedules that P is executed by a speed-controlled scheduler, under which the relative speed between any pair of $t_i, t_j \in T$ is $\gamma_i : \gamma_j$ in a unit-time interval (an epoch). For a sufficiently large epoch size $L < |\tau|$, as the trace τ becomes longer, we have

$$\lim_{L, |\tau| \to \infty} \frac{\left|(\tau_{k:k+L})\restriction_{t_i}\right|}{\left|(\tau_{k:k+L})\restriction_{t_j}\right|} = \frac{\gamma_i}{\gamma_j}$$

for any $1 \le k < |\tau| - L$.

Thread speed control can be implemented by dividing the program execution into slices by measuring real time, CPU cycles, or Lamport's logical clock [20]. A speed-controlled scheduler enforces epoch-based thread executions:

(1) $t_i \in T$ either executes all operations in a slice, or is blocked and executes Nop events).
(2) The ratio of slices received by t_i and t_j is roughly $\gamma_i : \gamma_j$ in an epoch (a period of time).

The idea of slicing the program execution in an epoch is first used to make a multithreaded program deterministic (aka. deterministic multithreading) [10], in which slices are executed under deterministic rules [27]. The similar technique can be used to execute P under any given speed vector γ.

For example, speed vector $\gamma = [0.2, 0.1, 0.2]$ may result in schedules $\langle t_1, t_2, t_1, t_3, t_3\rangle$ or $\langle t_1, t_3, t_2, t_3, t_1\rangle$. Both exactly match γ. In practice, $|\tau\restriction_{t_1}| : |\tau\restriction_{t_2}|$ may slightly differ from $\gamma_i : \gamma_j$, however, as long as threads roughly run at their designated speeds, the purpose of manifesting diverse thread schedules for testing multithreaded programs can be fulfilled. We also explicitly require $\gamma_i > 0$ because allowing $\gamma_i = 0$ permanently blocks t_i and leads to undefined ratios. We use ε to represent an extreme value near $\gamma_i = 0$.

Analogous to the interleaving space, the speed space Γ contains all possible speed vectors, i.e., $\Gamma = (0, 1]^n$. The rationale of introducing the speed space is discussed as follows.

3.3 Discussions

The speed space, an alternative representation of the interleaving space, is an *approximately continuous* space (thus is easy to sample) in which each vector corresponds to a particular *speed-controlled thread scheduler* and *preserves certain categories of concurrency bugs* (thus is useful in testing). We discuss these features as follow.

3.3.1 An Approximately Continuous Space. As mentioned in Section 2.3, the interleaving space of a multithreaded program is usually represented by a tree: a path from the root to a leaf denotes the schedule $\text{sched}(\tau)$ of a program execution τ. As for the motivating example, one thread in {`Main`, `Event`, `Verify`} is chosen to execute for one step at each schedule point, as shown in Figure 1a. This interleaving space is precise and complete (contains all possible schedules), but its discrete nature makes it difficult to sample.

On the other hand, the speed space is much easier to sample: one just picks up a γ value in the *hypercube* (Figure 1b) of the speed space Γ, and executing P under γ with a speed-controlled scheduler produces a trace τ. Various probability distributions can be easily defined over a hypercube, compared with a discrete tree. For example, we can sample schedules where particular threads run excessively fast/slowly to yield many extreme-case schedules. Effectively sampling of the speed space for testing multithreaded programs is later discussed in Section 3.4.

Note that different execution traces $\tau_1, \tau_2 \in \Sigma$ may correspond to the same speed vector $\gamma \in \Gamma$, as long as the threads run in a similar relative speed. Furthermore, there may be $\tau \in \Sigma$ which no speed vector could manifest. This is intentional because we would like to constrain threads' execution speeds at a coarse-grained level (each thread is controlled by a numeric value), and the interleaving space is reduced for easier sampling.

3.3.2 Speed Vectors as Schedulers. A speed vector corresponds to a *scheduler* with particular speeds (priorities) assigned to threads: for threads t_i and t_j, they run approximately at a relative speed of $\gamma_i : \gamma_j$. Therefore, the speed space defines *a family of schedulers* to manifest a diverse range of schedules. Many existing schedulers are related to speed vectors:

A native fair scheduler. The native operating system's scheduler $\mathcal{S}_{nfs}$ attempts to guarantee the fairness between threads. If not otherwise specified, each thread executes for an equal amount of CPU time in an epoch. $\mathcal{S}_{nfs}$ executes threads with the speed vector $[\frac{1}{n}, \ldots, \frac{1}{n}]$ for n threads.

The PCT scheduler for concurrency bug manifestation. $\mathcal{S}_{pct}$ assigns each thread with a random priority and chooses a thread with the highest priority to execute. At some priority-changing points, it changes an executing thread's priority to a lower value. To state briefly, $\mathcal{S}_{pct}$ executes threads with speed vector $[\varepsilon, \ldots, \gamma_i = 1, \ldots, \varepsilon]$ initially, and then changes it to $[\varepsilon, \ldots, \gamma_j = 1, \ldots, \varepsilon]$.

3.3.3 Concurrency Bugs in the Speed Space. The most important property of the speed space is that it guarantees to preserve many categories of concurrency bugs.

Most non-deadlock concurrency bugs in practice are either an *atomicity violation* in which two events that should happen atomically in a thread are interrupted, or an *order violation* in which the orderings between two events executed by different threads are

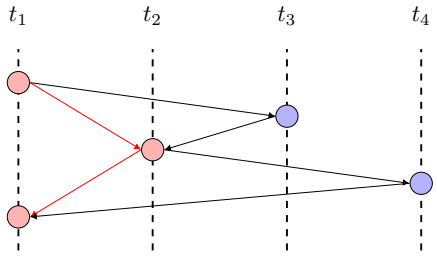


Figure 3: A bug example consisting of an A-B-A atomicity violation pattern (red) and two event orderings (blue). This bug has a depth of $d = 4$ and is preserved in the speed space.

inversed [22]. An order violation can be manifested by specifying the speeds of two threads where one thread runs faster than the other. An atomicity violation can also be manifested by setting correct speed vectors. Suppose that the atomicity of events a and b in thread t_i may be violated by another event c in thread t_j. To trigger the violation, we should ensure that

$$\frac{\pi_a}{\gamma_i} < \frac{\pi_c}{\gamma_j} < \frac{\pi_b}{\gamma_i}$$

where π_e is the number of events performed in $\tau\restriction_{e.t}$ before e. In other words, if these two threads' execution speeds satisfy

$$\frac{\pi_a}{\pi_c} < \frac{\gamma_i}{\gamma_j} < \frac{\pi_b}{\pi_c},$$

the atomicity violation can be manifested. An example of concurrency bug patterns being preserved in the speed space is shown in Figure 3, which resembles the event ordering in the previously unknown bug we found in the motivating example (Figure 2b). Such bugs can be easily manifested by speed-controlled schedulers under correctly set speed vectors, but are extremely difficult to be manifested by existing techniques.

More specifically, the speed space can preserve a more general A-B-A atomicity violation case with a bug depth of $d = 2(n-1)$, where a thread pair perform three A-B-A events and each of the other threads enforces one additional event in a particular order. Therefore, though the speed space is a *lossy reduction* of the interleaving space, effectively sampling it has the potential of manifesting real-world concurrency bugs [22].

3.4 Systematic Sampling of the Speed Space

We propose an algorithm for sampling the speed space, which is inspired by the idea of *exponential backoff* in computer networking: one waits for $2^0, 2^1, \ldots, 2^k$ units of times before the next attempt to obtain the exclusive access of a contented resource. Similarly, we systematically enumerate all thread pairs (t_i, t_j) and try all possible "backoffs" between t_i and t_j, as described in Algorithm 1. Threads t_i and t_j are called the *basis* which are systematically controlled using the following speed vectors:

(1) $\gamma_i = 2^{-(k+1)}$ and $\gamma_j \in V = \{2^{-k}, \ldots, 2^{-1}\}$;
(2) $\gamma_i = 1$ and $\gamma_j \in V = \{2^{-k}, \ldots, 2^{-1}\}$.

Algorithm 1: Systematic speed control

```
Input: Program P and threads T
Var: γ – speed vector
Var: V = {2^-k, ..., 2^-1} – speeds to assign
for (t_i, t_j) ∈ T × T and i ≠ j do
    for (v_1, v_2) ∈ {2^-(k+1), 1} × V do
        γ ← random speed vector
        (γ_i, γ_j) ← (v_1, v_2)
        Execute-with-Speed(γ)
```

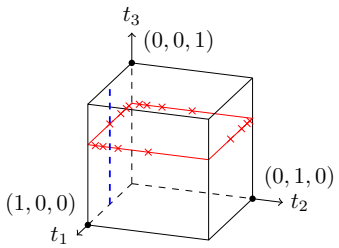


Figure 4: Sampled speed vectors (red marks) under the basis of (t_1, t_2) in the speed space Γ. The speed of t_3 is fixed to be γ_3, which is randomly chosen in $(0, 1]$ (the dashed blue line).

The speeds of other threads are set uniformly at random in $(0, 1]$. In other words, the speed value γ_i is set to either maximum (1) or minimum ($2^{-(k+1)}$), while γ_j enumerates all speed values of 2^{-k}.

The speed space sampling algorithm is better geometrically explained in the hypercube of $\Gamma = (0, 1]^n$: fixing the basis (t_i and t_j) yields a sub-cube of dimension $n-2$ which is sampled uniformly at random. Considering a multithreaded program with $T = \{t_1, t_2, t_3\}$ and (t_1, t_2) are set as the basis, sampled points are shown in Figure 4. For $\gamma = [\gamma_1, \gamma_2, \gamma_3] \in \Gamma$, points (γ_1, γ_2) are always on the boundary of the t_1-t_2 plane, and γ_3 is sampled uniformly at random (illustrated by the blue dashed segment in the figure). In the more general case where the basis is not fixed, all six surfaces of the cube (Γ) are sampled in a similar way.

Such a sampling algorithm enables us to exercise various kinds of corner-case schedules, resulting in the manifestation of concurrency bugs oftentimes missed in thread executions under a native scheduler: $\mathcal{S}_{nfs}$ is just a single point in the diagonal line[4]. On the other hand, for each pair of basis threads (t_i, t_j), a broad spectrum of $\gamma_i : \gamma_j$ are exercised:

$$2^{-k}, 2^{-(k-1)}, \ldots, 2^{-1}, 2^1, 2^2, \ldots, 2^k,$$

plus other threads having a diverse range of running speeds. We intentionally do not sample $\gamma_i : \gamma_j = 1$ (2^0) because it is the default behavior of $\mathcal{S}_{nfs}$.

[4] All speed vectors in the form of $\alpha \cdot [1, 1, \ldots, 1]$ are equivalent. However, Algorithm 1 never samples two equivalent speed vectors.

Algorithm 2: Speed-controlled scheduler

```
Input: T – set of total threads
Input: γ – speed vector
Var : W – waiting list
Var : L – number of total events to execute in an interval
1  Procedure Execute-with-Speed(γ)
2      for each interval do
3          parallel-for t_i ∈ T do
4              Schedule(i, γ_t, 0)
5          W ← ∅
6          ▷ signal (wake up) all threads in W
7  Procedure Schedule(i, γ_i, d)
8      if d/L = γ_i then
9          W ← W ∪ {t_i}
10         ▷ wait for signal
11     else
12         ▷ execute t_i for one step
13         Schedule(i, s, d + 1)
```

4 SCHNAUZER

In this section, we describe our prototype tool–Schnauzer[5], which combines the speed-controlled scheduler and trace-based analysis to test multithreaded programs effectively.

Schnauzer consists of three components: a speed space sampling module, a *speed-controlled scheduler* for executing program P with a speed vector $\gamma \in \Gamma$ to obtain an execution trace τ, and a *trace-based analyzer* for predicting bugs/anomalies based on τ.

4.1 The Speed-controlled Scheduler

The speed-controlled scheduler $\mathcal{S}_{scs}$ is described in Algorithm 2. Procedure `Execute-with-Speed` is called with speed vector γ to schedule threads in T. It divides a program's execution into intervals with a predefined length L, which denotes the expected number of events to be executed in an interval.

In each interval, the procedure `Schedule` coordinates the execution of all threads. `Schedule` maintains d, a thread-local counter of the number of events that t have already executed in this interval. When thread t has drained its quota ($\frac{d}{L} = \gamma_t$), t is added to a waiting list (denoted by W) and is blocked until the next interval.

Note that we execute a thread t regardless of whether it is blocked. If t is blocked (e.g., waiting for a mutex lock, or pending on a condition variable), executing t yields an Nop event. Since we require that $\gamma_i > 0$ for all i (i.e., threads have positive speeds), there must be at least one thread executing a non-Nop event in an interval, i.e., the process does not stall.

4.2 Trace-based Analysis

Given an execution trace, trace-based analyses predict the existence of potential concurrency bug patterns in other execution traces. For example, data race[6] is one of the most common causes of real-world concurrency bugs [22]. Even if an execution trace does not have two consecutive racing events, trace-based analyses may *predict* the existence of such anomalies.

To predict data races in a multithreaded program, one may use the *happens-before* relation $\prec_{hb}$ [24]. Two events $e_i \prec_{hb} e_j$ if either of the following conditions holds, assuming that e_i and e_j are the i-th and j-th events in τ, respectively:

(1) $i < j$ and $e_i.t = e_j.t$, i.e., e_i happened before e_j in the same thread;
(2) $i < j$ and e_i and e_j are synchronization operations performed on the same variable;
(3) e_i and e_j are transitively dependent using the above two rules.

A pair of events (e_i, e_j) is a happens-before race if $e_i.x = e_j.x$, $e_i.t \neq e_j.t$, at least one is a write, and they are not ordered in $\prec_{hb}$.

Happens-before race detectors [4, 13, 25] are often simple but limited due to their overly conservative HB relation construction, which forms extra happens-before relations. *Causally-precedes* (CP) [19, 36] soundly relaxes the causal model in order to improve the detection power. Under the relaxed model, it does not establish HB relations between critical sections that have no conflicting accesses. *Predictive trace analysis* (PTA) [16, 17] soundly reorders events in an execution trace by SMT (satisfiability modulo theories) solvers [9], which can predict concurrency faults unseen in the input trace.

Though both CP and PTA can predict potential concurrent bugs in a family of schedules (not only in τ), the successful prediction of a particular concurrency bug pattern heavily depends on the quality of the input execution trace [19]. We found that our speed-controlled scheduler (which excels in exercising adversarial/abnormal schedules) is orthogonal to trace-based analyses. In this way we combine these two techniques: we first sample a series of diverse thread schedules, and then these schedules are further checked against concurrent bug patterns using a happens-before race detector.

4.3 Implementation

We described in Section 3 that the speed-controlled scheduler controls execution speeds of threads at a fine-grained level: it restricts the number of *events* a thread can execute in an interval. In practice, such a fine-grained control incurs large and unnecessary overhead. A recent research [18] showed that most real-world concurrency bugs can be manifested by a *coarsely interleaved* scheduler. The coarse-interleaving hypothesis motivated us to conduct speed control at a coarse-grained level to reduce the overhead, and diverse schedules can still be sampled. Therefore, in our Schnauzer implementation, thread t_i receives an upper limit on the number of *function calls* proportional to γ_i in an interval.

We also found that a master-worker pattern is widely used in real-world programs where a main thread usually creates worker threads to perform similar tasks. This observation implies that there is no need to exhaustively enumerate all thread pairs (t_i, t_j) as basis (in Algorithm 1). Enumerating the basis in a small set of $T_s \subseteq T$ consisting of the master thread and a few workers suffices for effectively sampling of the interleaving space.

We implemented Schnauzer on top of LLVM [2]. Schnauzer instruments LLVM IR instructions to trace Read and Write events,

[5] Available at: https://midwinter1993.github.io/Schnauzer/
[6] Two consecutive events e_1 and e_2 in τ where $e_1.x = e_2.x$ and Write $\in \{e_1.op\} \cup \{e_2.op\}$.

Table 1: Real-world multithreaded programs with previously known bugs used in the evaluation.

Program	Description	LOC	#Thread
Aget	file downloading	2k	4
Cherokee	Web server	80k	4
Httpd	Web server	290k	4
Mysql	relational database	390k	15
Transmission	bit-torrent client	87k	4
Pbzip2	parallel compression	2k	3

Table 2: Happens-before races detected by SCS and PCT.

Program	#Data Races		Improvement
	SCS	PCT	
Aget	4	4	0 (+**0**%)
Cherokee	191	70	121 (+**172**%)
Httpd	1,272	1,316	-44 (-**3**%)
Mysql	60	46	14 (+**30**%)
Transmission	71	55	16 (+**29**%)
Pbzip2	24	22	2 (+**9**%)

and inserts a callback to the Schnauzer before each function is called for speed control. This mechanism also produces a log of synchronization operations, which is obtained by logging function calls into the `pthread` library.

A speed-controller thread maintains the running status, statistics, and quota of each application thread (including a dynamically created one). $\prec_{hb}$ data races are detected by maintaining and comparing a pair of memory access events' vector clocks.

5 EVALUATION

The evaluation of our SCS scheduler focuses on answering the following two research questions:

RQ1: Can SCS sample diverse schedules of multithreaded programs in limited testing budgets?

RQ2: Is Schnauzer effective in testing multithreaded programs?

Both questions are difficult to answer directly. Therefore, we take an indirect approach that adopts bug-related quantitative metrics to compare SCS with PCT.

The first question is answered by using quantitative indicators of the testing thoroughness on a set of popular open-source projects that are mature and widely deployed, which have also been extensively tested and studied in the previous work [29, 35, 37, 38, 40]:

(1) whether previously *known* concurrency bugs can be successfully manifested;
(2) the number of statement pairs s_1, s_2 that are unordered by happens-before ($\prec_{hb}$) if these known bugs are fixed.

The second question is answered by applying Schnauzer (both the speed-controlled scheduler and a happens-before race detector) to the same set of programs to find whether it can reveal *previously unknown* concurrency bugs that require complex schedules to trigger.

5.1 Experimental Setup

5.1.1 Subjects. The multithreaded programs used in the evaluation are shown in Table 1. We evaluated the SCS scheduler on a set of real-world multithreaded programs of varying complexity used by previous work [37]. Each subject contains a known concurrency bug and a patch for fixing it.

5.1.2 Quantitative Measure of Schedule Diversity. We use the number of *happens-before race statements* in the programs' execution traces to reflect the degree of schedule diversity. A pair of program statements (s_1, s_2) is a happens-before race in an execution trace τ if and only if there exist events $e_1, e_2 \in \tau$ such that:

(1) e_1 and e_2 are produced by executing s_1 and s_2, respectively.
(2) (e_1, e_2) is a happens-before race: both events accessed the same shared variable (i.e., $e_1.x = e_2.x$), at least one is a write (i.e., Write $\in \{e_1.op\} \cup \{e_2.op\}$), and

$$\neg(e_1 \prec_{hb} e_2) \wedge \neg(e_2 \prec_{hb} e_1).$$

If a technique sampled more happens-before race statement pairs (s_1, s_2), we consider that the technique is more effective in generating diverse schedules. This is because even if s_1 and s_2 may simultaneously happen under a particular thread schedule, these two statements may not even appear in other schedules, or there may exist synchronization operations in between to make the events being ordered in $\prec_{hb}$. Intuitively, closer the events produced by s_1 and s_2 are scheduled, larger the chance they are forming a happens-before race [13]. Section 5.3 presents more discussions on this quantitative measure.

5.1.3 Experimental Setup. We compared SCS with PCT [5], the state-of-the-art interleaving space sampling technique with probability bug-finding guarantee. For each program, we supplied it with a simple test input that simulates a daily use case of the program. We used apache ab [31] to send the same set of requests to Httpd and Cherokee, both serving simple static contents; Aget and Transmission downloaded an online file of 1GB size; Pbzip2 decompressed a file of 1M size; Mysql executed a script that drives two threads to simultaneously insert data into a table.

In the experiments, we assume that all subject programs receive a similar amount of testing resource budgets. Therefore, we choose $T_s \subseteq T$ to be the main (master) thread and two worker threads (therefore $|T_s| = 3$), and exhaustively numerate all basis thread combinations in T_s. Choosing a fixed set of T_s is effective in testing because all evaluated programs follow the master-worker concurrency pattern, and an existing study [22] shows that most concurrency bugs can be manifested within three threads.

The back-off parameter is set to $k = 7$ for an approximately 100× maximum speed difference in the speed-controlled scheduler (relative speeds range from 1/2 to 1/128). We limit k to a small constant because a relative speed is an exponential of k, and a large k would deprive a slow thread of being executed.

There are

$$\binom{|T_s|}{2} \cdot 2k$$

distinct speed vectors sampled for each evaluated subject program, and this value equals 42 in our experimental setting. For PCT, we set the bug depth $d = 3$ as recommended in the authors' original

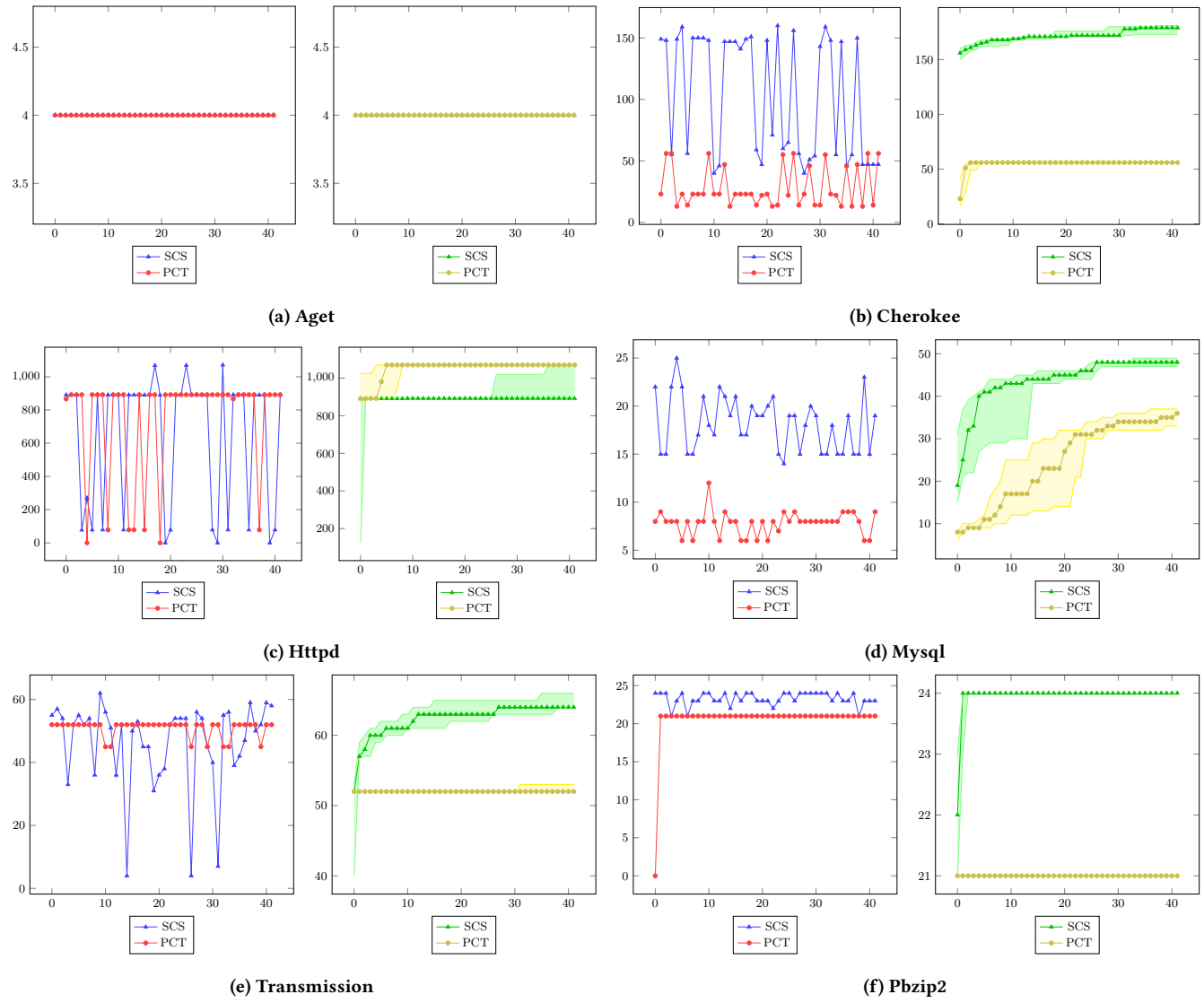


Figure 5: Distinct race statement pairs found by SCS and PCT. The left plot (blue/red) of each subject displays the number of distinct race statement pairs in a round of 42 program executions (for SCS, executions are scheduled under different speed vectors; for PCT, priority changes at random points). The right plot (green/yellow) of each subject displays the medium of cumulated distinct race statement pairs, shaded area denoting the 25th–75th percentile.

paper [5], and sample the same amount of schedules (42 random runs) for a fair comparison.

In answering RQ1, we first run each subject program (with a previously known concurrency bug and assertions to detect it) under the 42 sampled speed vectors to see if all concurrency bugs can be manifested. Then for the quantitative evaluation, we patch all known concurrency bugs to avoid early program crashes, and run both SCS and PCT to collect race statement pair statistics.

In answering RQ2, we used the latest versions of the subject programs in which all known concurrency bugs were fixed, and run each program under the 42 sampled speed vectors. We examine the crash or data race reports to find if there is previously unknown bugs in the latest versions.

All experiments were conducted on a desktop with a quad-core 3.40GHz Intel Core i7 processor and 16GB memory running Ubuntu Linux 16.04. In answering RQ1, both SCS and PCT were executed for 10 independent testing rounds and statistical data is collected. In answering RQ2, we only run each subject program once to simulate a typical in-house testing scenario.

5.2 Experimental Results

5.2.1 Answering RQ1 (Schedule Diversity). First, all known concurrency bugs can be successfully manifested by SCS: assertions are

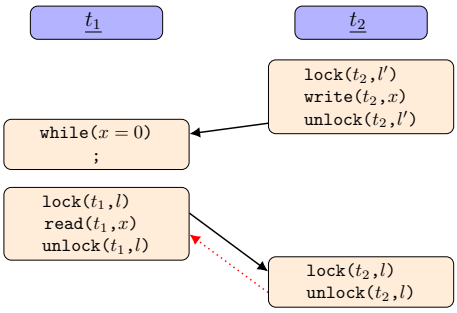


Figure 6: A typical thread schedule found by a speed-controlled scheduler but not by PCT.

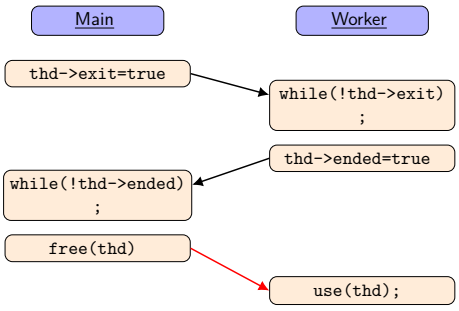


Figure 7: Illustration of a previously unknown bug in the Cherokee Web server. The event ordering in red triggers a concurrent use-after-free bug.

violated in running all subjects. The overall quantitative experimental results are shown in Table 2. The number of distinct race statement pairs (s_1, s_2) in all 10 rounds found by SCS and PCT are displayed in Columns 2 and 3, respectively.

Summarizing the results, SCS found 109 (7.2%) more pairs of unique race statements compared with PCT. Excluding Aget, which adopts a trivial concurrency pattern (therefore SCS and PCT found the same set of race statement pairs), SCS outperforms PCT on Cherokee (+172%), Mysql (+30%), Transmission (+29%), and Pbzip2 (+9%). Though it found 44 (-3%) less pairs than PCT on Httpd, it still has a better overall result.

The detailed statistics of race statement pairs (RSes) are shown in Figure 5. In each figure, the horizontal axis denotes the number of program runs and the vertical axis denotes the number of distinct RSes. In each of the 10 testing rounds, 42 traces $\tau_1, \tau_2, \ldots, \tau_{42}$ are obtained. The left sub-figure (blue/red) of each subject displays the number of distinct RSes in a randomly chosen testing round of PCT and SCS, in which a point (i, n) denotes that a technique found n distinct RSes in τ_i. The right sub-figure (green/yellow) of each subject displays the distribution of cumulated RSes, in which a point (i, n) denotes that a technique found n distinct RSes in execution traces $\tau_1, \tau_2, \ldots, \tau_i$. The shaded area shows the 25th–75th percentile over the 10 rounds of our experiment.

The right sub-figures (green/yellow plots) show that SCS detected more data races for programs as the growth of sampled schedules, except for Aget and Httpd. Aget is a small-scale application with a simple concurrency pattern. Both SCS and PCT detected the same set of data races, which we believe are all possible happens-before races in the program. For Httpd, though SCS detected less RSes, the third quantile indicates that SCS is capable of detecting almost the same data races as PCT in many testing rounds. For the remaining subjects, SCS has a better trend in detecting data races as more schedules are sampled, particularly on Cherokee, Mysql, and Transmission.

Figure 6 shows a typical schedule from Pbzip2 sampled by SCS but not by PCT. If one attempts to sample this schedule by PCT, one should first set t_2 with a higher priority; otherwise, t_1 will always spin on variable x. After t_2 writes x, it must be changed to a lower priority, thus, letting t_1 exit the loop and acquire the lock l. However, we see that the priority-changing point for t_2 is delicate: if the priority of t_2 is not changed immediately, it will run continuously, resulting the HB edge represented by the red line. SCS can sample this schedule more easily: if t_1 runs faster than t_2, t_2 will write x when t_1 is spinning, and then, t_1 may acquire the lock earlier than t_2 because it runs faster.

5.2.2 Answering RQ2 (Testing Effectiveness). To our surprise, applying Schnauzer to the latest versions of the subjects found previously unknown concurrency bugs: Schnauzer directly crashed the Transmission Client, and found a rarely occurring race in Cherokee Web server. In total, we found three previously unknown bugs in the evaluated subjects, and that all bugs were buried a long time before Schnauzer revealed them. For example, in Cherokee the bug was traced back to the 1.0 version, even before the project was migrated to Github.

The two bugs in Transmission, including the motivating example, are similar but in different components of the program. The triggering of these two bugs are discussed in Section 2. The manifestation of the concurrency bug in Cherokee is shown in Figure 7. Under a typical native fair scheduler, the `main` thread tends to wait for the `worker` thread to exit before the `main` thread reclaims memory resources. However, in rare cases, an atomicity violation (between Line 4 and Line 5 of the `worker` thread) can be manifested, resulting in a concurrent use-after-free bug. We also proposed a fixing suggestion to its developers.

We believe that other concurrency testing work also extensively tested these subjects, including Transmission and Cherokee, under various settings. However, our speed-controlled scheduler is the first to find the bugs. Therefore, we believe that speed space sampling is promising in effectively testing real-world multithreaded programs.

5.3 Threats to Validity

A key external threat to validity is that data races may not reflect the diversity of a schedule. Furthermore, a pair of racing events (e_1, e_2) unordered in $\prec_{hb}$ may not imply that these two events could concurrently happen: the soundness of a $\prec_{hb}$ race detector only holds for the *first* pair of unordered events, as for all subsequent racing events may not even happen. For instance, in Figure 6, Read(t_1, x) and Write(t_2, x) are detected as a $\prec_{hb}$ race, however, these two

statements could never be executed in parallel (and thus is not a data race) because t_1 spins on x in between.

Nevertheless, suppose that a pair of events (e_1, e_2) unordered by $\prec_{hb}$ can be detected by SCS but not PCT, it implies that either of the following two conditions holds:

(1) e_1 and e_2 do not simultaneously occur in all execution traces sampled by PCT, or
(2) e_1 and e_2 are always ordered by $\prec_{hb}$ in all execution traces sampled by PCT.

In other words, SCS has found a particular schedule that both e_1 and e_2 are manifested and they are made closer (and thus not ordered by $\prec_{hb}$). Consequently, even if the race detector may not be sound, we believe that the technique that can find more distinct race statements is more effective in sampling diverse schedules of multithreaded programs.

6 RELATED WORK

Much prior work has focused on effectively exercising (sampling) buggy schedules in the interleaving space for testing a multithreaded program. We categorize these techniques into model checking, random sampling, or heuristic sampling. We also discuss trace-based analyses in this section.

Model Checking. The goal of model checking [12] is to prove (or disprove) that all schedules in the interleaving space satisfy a certain correctness criterion. Model checking is intractable due to the exponential interleaving space, and techniques have been proposed to soundly reduce the search space to be checked [7, 15, 39]. Despite these efforts, conducting full model-checking on large real-world multithreaded programs are still considered impractical.

Therefore, bounded model-checking, in which only a *subset* of the interleaving space is checked, becomes a natural solution. Based on the fact that many concurrency bugs can be manifested by simple thread interleaving (e.g., by permuting a few events in a few threads), delay-bounding [11] and preemption-bounding [26] strike a nice balance between resource cost and testing effectiveness.

The speed space presented in this paper is also a lossy reduction of the interleaving space. Compared with the reduced interleaving space of a delay-bounded or preemption-bounded scheduler, the speed space is approximately continuous and thus is easier for us to sample. In particular, by sampling extremely valued vectors in the space, adversarial/abnormal schedules can be manifested by a speed-controlled scheduler.

Random Sampling. Randomly manipulating thread schedule to sample the interleaving space for manifesting concurrency bugs is usually lightweight and applicable in practice. Random sampling techniques may insert random sleeps, execution delays, or thread suspensions at synchronization points [33] or, reverse thread priorities at random program points.

Random samples may be effective in testing with probabilistic bug-triggering guarantee [5]. However, there may also be specific event orderings (e.g., the thread schedule to trigger the bug in Figure 2) whose manifestation is of an extremely low probability. This is partly due to the discrete nature of the interleaving space, and it is a challenge in designing probability distributions for effective sampling of schedules that may relate to concurrency bugs.

The speed space offers new opportunities for designing effective random sampling techniques toward adversarial/abnormal schedules. Our speed space sampling algorithm (Algorithm 2) is based on the uniform distribution sampling in a hypercube.

Heuristic Sampling. Heuristics facilitate effective interleaving space sampling techniques. Concurrency coverage criteria [6, 21], concurrency bug patterns [22], and commutativity of event orderings [34] helped the sampler avoid wasting time manifesting schedules which are less likely to be buggy.

Our speed space sampling algorithm also adopts heuristics to find adversarial/abnormal schedules. Particularly, we systematically exercise all thread basis pairs and uniformly sample schedules at random in the $n-2$ dimension space for diverse schedules, which attempts to avoid sampling similar schedules.

Trace-based Analysis. A trace-based analysis takes an execution trace τ as its input, and verifies a certain property Φ against a family of schedules $\mathcal{F}(\tau)$, i.e., checking if there is $\tau' \in \mathcal{F}(\tau)$ such that $\Phi(\tau')$ holds. $\mathcal{F}$ is usually a causal model of events under a particular memory model [17, 19, 36], while Φ can be a detector of data races [13], atomicity violations [23], or other types of concurrency bugs [16].

Interleaving space sampling algorithms (e.g., delay-bounded sampling [11], preemption-bounded sampling [26], and Algorithm 1) are orthogonal to trace-based analyses. A trace-based analysis can successfully predict a concurrency bug only if bug-related events are manifested (and usually close enough) in the execution trace. Effective sampling algorithms can help trace-based analysis detect more subtle concurrency bugs.

7 CONCLUSION

This paper presents the definition, sampling algorithm, and application in data race detection of the *speed space*, an alternative representation of a multithreaded program's interleaving space. By systematically sampling speed vectors in the speed space and executing a multithreaded program under a speed-controlled scheduler, adversarial/abnormal schedules can be manifested for detecting concurrency bugs. The evaluation of our prototype Schnauzer shows promising results in effective testing of multithreaded programs, with previously unknown concurrency bugs in real-world programs being detected.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for comments and suggestions. This work is supported in part by National Key R&D Program (Grant #2018YFB1004805) of China, National Natural Science Foundation (Grants #61690204, #61802165) of China, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] [n. d.]. Cherokee. http://cherokee-project.com/.
[2] [n. d.]. The LLVM compiler infrastructure. http://llvm.org/.
[3] [n. d.]. Transmission. https://transmissionbt.com/.
[4] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

[5] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. 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)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1736020.1736040

[6] Yan Cai and Lingwei Cao. 2015. Effective and Precise Dynamic Detection of Hidden Races for Java Programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. ACM, New York, NY, USA, 450–461. https://doi.org/10.1145/2786805.2786839

[7] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158119

[8] Ankit Choudhary, Shan Lu, and Michael Pradel. 2017. Efficient Detection of Thread Safety Violations via Coverage-guided Generation of Concurrent Tests. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 266–277. https://doi.org/10.1109/ICSE.2017.32

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[10] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1508244.1508255

[11] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. 2011. Delay-bounded Scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 411–422. https://doi.org/10.1145/1926385.1926432

[12] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. Con2Colic Testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. ACM, New York, NY, USA, 37–47. https://doi.org/10.1145/2491411.2491453

[13] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[14] Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

[15] Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 165–174. https://doi.org/10.1145/2737924.2737975

[16] Jeff Huang, Qingzhou Luo, and Grigore Rosu. 2015. GPredict: Generic Predictive Concurrency Analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 847–857. http://dl.acm.org/citation.cfm?id=2818754.2818856

[17] Jeff Huang, Patrick Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'14)*. ACM, 337–348. https://doi.org/10.1145/2594291.2594315

[18] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 582–598. https://doi.org/10.1145/3132747.3132767

[19] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

[20] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[21] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. 2007. A Study of Interleaving Coverage Criteria. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers (ESEC-FSE Companion '07)*. ACM, New York, NY, USA, 533–536. https://doi.org/10.1145/1295014.1295034

[22] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[23] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. 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)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1168857.1168864

[24] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

[25] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/1542476.1542491

[26] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 446–455. https://doi.org/10.1145/1250734.1250785

[27] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/1508244.1508256

[28] Chandandeep Singh Pabla. 2009. Completely Fair Scheduler. *Linux J.* 2009, 184, Article 4 (Aug. 2009). http://dl.acm.org/citation.cfm?id=1594371.1594375

[29] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. 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)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[30] PCWorld. 2012. Nasdaq's facebook glitch came from race conditions. http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html.

[31] Apache Server Project. [n. d.]. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[32] SecurityFocus. 2004. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

[33] Koushik Sen. 2007. Effective Random Testing of Concurrent Programs. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 323–332. https://doi.org/10.1145/1321631.1321679

[34] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 11–21. https://doi.org/10.1145/1375581.1375584

[35] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do I Use the Wrong Definition?: DeFuse: Definition-use Invariants for Detecting Concurrency and Sequential Bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 160–174. https://doi.org/10.1145/1869459.1869474

[36] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

[37] Jie Yu and Satish Narayanasamy. 2009. A Case for an Interleaving Constrained Shared-memory Multi-processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 325–336. https://doi.org/10.1145/1555754.1555796

[38] Mingxing Zhang, Yongwei Wu, Shan Lu, Shanxiang Qi, Jinglei Ren, and Weimin Zheng. 2014. AI: A Lightweight System for Tolerating Concurrency Bugs. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, New York, NY, USA, 330–340. https://doi.org/10.1145/2635868.2635885

[39] Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic Partial Order Reduction for Relaxed Memory Models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 250–259. https://doi.org/10.1145/2737924.2737956

[40] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 251–264. https://doi.org/10.1145/1950365.1950395