ABSTRACT

Finding buggy schedules in a multithreaded program’s interleaving space is challenging because of the astronomically large space. To address the issue, the state-of-the-art techniques either enumerate all schedules in a bounded space or sample schedules by heuristics. Though they revealed bugs in practical systems, they may still be ineffective in detecting complex concurrency bugs. In this paper, we propose a new approach to controlling the execution speed of threads and project the discrete interleaving space into a speed space which is approximately continuous and easy-to-sample. Based on the new space, we sample schedules systematically to explore adversary/abnormal schedules. We also cascade the trace-based analysis to our scheduler, implementing a prototype testing tool. The experiments confirm that our speed control scheduler is able to sample diverse schedules with more data races compared to the state-of-the-art interleaving sampling techniques. We used the prototype tool in practice and found previous unknown concurrency bugs in open source programs.

CSC CONCEPTS

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

KEYWORDS

Concurrency; testing; interleaving space sampling; race detection

1 INTRODUCTION

Multithreaded programs are prevalent but also error prone. Concurrency bugs have caused serious real-world consequences, e.g., the Northeast blackout [34] and mismatched NASDAQ Facebook share prices [32].

Unlike sequential bugs, the manifestation of concurrency bugs depends not only on inputs, but also on thread schedules. So we often test a multithreaded program by exercising different schedules in its interleaving space, in hope that hidden concurrency bugs will be manifested. However, the size of a multithreaded program’s interleaving space (all possible schedules when program inputs are fixed) is exponential of its execution steps. 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 exercises schedules randomly [5], by heuristics [36] or in a bounded interleaving space [28]. However, these existing approaches may incur wastes 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 and easy-to-sample. It helps us exercise adversary/abnormal schedules more easily and develops effective testing techniques. Our approach exploits the following key observations:

• Events usually happen in a particular order under a native scheduler because threads run with almost the same “execution speed” [30].
• Events leading to many concurrency bugs in real systems are coarsely interleaved [20].
• Interleaving space sampling and predictive analysis [13, 21] complement each other.

The first observation suggests that to effectively sample 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 point denotes a set of schedules whose threads run in certain relative speeds. This projection not only yields a reduced space but also makes sampling of adversarial/abnormal thread schedules easier (in Figure 1c).

The second observation, also known as the coarse interleaving hypothesis [20], implies that the projected speed space can be sampled in a coarse-grained level. We found that the speed space can be further reduced by only considering function-call schedules instead of 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 [21] (in Figure 1d). A predictive analysis explores a larger set of schedules (usually defined by a causal-model [18]) in the interleaving space given a seed schedule. Feeding such an analysis with diverse seeds (samples in the interleaving space) helps us reveal long-time hidden concurrency bugs in real-world programs.

In summary, we test a multithreaded program by sampling its reduced speed space and cascading with 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 is able to explore schedules with more data races in most benchmarks, varying from 9% to 172%. Schnauzer also found
previously unknown concurrency bugs in Cherokee [1] and Transmission [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. 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 shows 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 whole interleaving space of a multithreaded program contains an astronomical number of schedules. Techniques for testing multithreaded programs have been proposed. For example, model checking for concurrency [12] attempts to enumerate all possible program states (and schedules) to find the buggy schedules. When the enumeration is intractable for large programs, windowing techniques [18, 40] and bounding techniques [11, 28] are widely used. The former attempts to divide the execution into a sequence of fixed-size windows and performing analysis on each window separately; whereas the latter bounds the state space to explore. On the other hand, PCT (Probabilistic Concurrency Testing) [5] is a random testing approach that can detect a concurrency bug of bug depth $d$ (i.e., the minimum number of specific event orderings sufficient to find the bug) with probability at least $1/nk^{d-1}$ in a multithreaded program with $n$ threads and $k$ execution steps. These tools are effective against finding concurrency bugs, and have revealed many bugs in practical systems.

2.2 Motivating Example

Consider a concurrency bug in the Transmission Client [3] as illustrated in Figure 2. The operating system’s native scheduler almost definitely schedules the request sent by the verify thread before the main thread as verifyDone is usually invoked at the beginning while sessionClose is called at the end, forming the event ordering in Figure 2a. In some occasions (Figure 2b), verifyDone can be scheduled earlier than sessionClose, yielding a use-after-free on tor. To manifest this bug, the main thread is required to send the sessionClose request before the verify thread and exit after the event thread. Three event orderings are required to manifest the bug: the main thread must be forced to send requests 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 bug depth $d = 3$, implying 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 infeasible for these practical programs with huge traces. Limiting the checked trace in a small window [18, 40] 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, there is no data race involved in the schedule (both free and use happen in the same thread), race-directed techniques [36] cannot capture the bug.

2.3 The Need for an Alternative Representation of the Interleaving Space

Finding these bug-triggering schedules in the huge interleaving space is a challenge. The most classical way to explore the interleaving space is to traverse its state transition tree in which a node

\[ t_1, t_2, t_3 \]

\[ (1, 0, 0) \]

\[ (0, 1, 0) \]

\[ (0, 0, 1) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]

\[ (1, 0, 0) \]

\[ t_1, t_2, t_3 \]

\[ (0, 0, 1) \]

\[ (0, 1, 0) \]
Main Event Verify
async(sessionClose)
eventClose(session)
verifyDone:
tor->member=...
sessionClose:
free(tor)
async(verifyDone)

(a) Typical event orderings under a native scheduler

async(sessionClose)

Main Event Verify
async(sessionClose)
eventClose(session)
verifyDone:
tor->member=...
sessionClose:
free(tor)
async(verifyDone)

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

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

\( u \) denotes a program state and an edge \((u, v)\) indicates that \( v \) is the state obtained by executing one step by a thread from \( u \), as shown Figure 1a. This is the basis of many (stateful or stateless) model checking techniques \([12, 16]\).

On the other hand, both PCT \([5]\) and the delay-bound scheduler \([11]\) ingeniously transform this general tree into a binary tree. For most of the state transitions, the two approaches simply proceed the current running thread, and perform context switches (or preemptions) at certain program points. The transformation yields both easier (a path/trace can be encoded as a bit string) and more effective (by observing that many practical bugs can be manifested with a few context switches, only a small subspace of a few bits is needed for sampling) sampling of the interleaving space to find concurrency bugs.

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 \( S \) 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 yields an event \( e \) of the following types:

- \( \text{Read}(t, x)/\text{Write}(t, x) \) for thread \( t \) accessing the shared memory variable \( x \).
- \( \text{Lock}(t, \ell)/\text{Unlock}(t, \ell) \) for thread \( t \) performing a synchronization operation on the lock variable \( \ell \).
- \( \text{Fork}(t, u)/\text{Join}(t, u) \) for thread \( t \) forking/joining a thread \( u \).
- \( \text{Nop} \) for doing nothing when thread \( t \) is blocked while being chosen to execute\(^2\).

We use \( e.\text{op} \in \{\text{Read}, \text{Write}, \text{Lock}, \text{Unlock}, \text{Fork}, \text{Join}\} \), \( e.t \in T \), and \( e.x \) to denote the operation type, the thread, and the variable associated with an event \( e \). Chronologically concatenating the events yields an execution trace \( \tau \), a sequence of events \( \tau = (e_1, \ldots, e_m) \). We denote events in \( \tau \) that are performed by thread \( t \) as \( \tau_t, \) and use \( \tau_{ij} = (e_i, e_{i+1}, \ldots, e_j) \) to denote a substring of \( \tau \). The schedule of a trace \( \tau \), \( \text{sched}(\tau) \), is defined as the sequence of thread identifiers in \( \tau: \text{sched}(\tau) = (t_1, t_2, t_3, \ldots, t_m) \).

The interleaving space \( \Sigma \) 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., \( |\Sigma| = O(m^n) \).

Concurrency bugs are triggered by particular event orderings: as long as we do not know which events may lead to bugs, manifesting such orderings seems to help find needles in a haystack. Even worse, given a prefix of an execution trace \( \tau \), it is intractable to predict the subsequent events in \( \tau \). These observations make sampling of \( \Sigma \) challenging: even if we have collected a set of traces in testing, the criterion for guiding the next schedule is unclear (and therefore existing techniques are mostly supported by heuristics \([5, 8, 11, 36]\)).

Keeping these two ideas in mind, we seek our representation of \( \Sigma \) such that schedules are easy to generate while concurrency bugs are still preserved for triggering.

3.2 The Speed Space

The idea that concurrency bugs are triggered by specific 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 \( \gamma \) represents schedules that \( P \) is ran by forcing the relative speed between any pair of \( t_i, t_j \in T \) to be \( \gamma_i : \gamma_j \) in a

\(^2\)We introduce \( \text{Nop} \) for an easier description of the semantics in a speed-controlled scheduler.
unit-time interval (an epoch). For a sufficiently large time interval $L < |\tau|$, as the trace becomes longer, we have

$$
\lim_{L \to \infty} \frac{\epsilon L}{|\tau|} = \frac{\epsilon}{\gamma_i} \Rightarrow Y_i = Y_j
$$

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

Speed control can be done by dividing the program execution into slices indicated by real time, CPU cycles, or Lamport’s logical clock [22]. Each slice is assigned to a thread $t_i \in T$ for execution (a blocked thread simply executes Nop events). We allocate slices to reflect the relative speed between threads: the scheduler enforces that in an epoch (a period of time), the ratio of slices received by $t_i$ and $t_j$ is roughly $y_i : y_j$. The idea of scheduling slices in an epoch is first used to make a multithreaded program deterministic (aka. deterministic multithreading) [10], which coordinates slices under deterministic rules [29]. We adopt the similar idea to execute $P$ under any given $\gamma$.

For example, a speed vector $\gamma = [0.4, 0.2, 0.4]$ may result in schedules $(t_1, t_2, t_1, t_3, t_3)$ or $(t_1, t_2, t_2, t_3, t_1)$ that exact match $\gamma$. In practice, $|\tau_{t_1}| : |\tau_{t_2}|$ may slightly differ from $y_1 : y_2$, however, as long as threads roughly run at the designated speeds, the purpose of testing can be fulfilled. We also explicitly require $y_i > 0$ because allowing $y_i = 0$ essentially blocks $t_i$ permanently, and may lead to undefined ratios. We use $\epsilon$ to represent an extreme value near $y_i = 0$.

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

### 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 point corresponds to a thread scheduler and preserves certain kinds of concurrency bugs (thus can find concurrency bugs). These features are discussed in the following.

#### 3.3.1 An Approximately Continuous Space

Conventionally, the interleaving space of a multithreaded program is regarded as a tree: at each node, one thread is chosen to execute and a path from the root to a leaf is a complete schedule for one execution. As for the motivating example, one of three threads is chosen to execute repeatedly at each step, 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 $\gamma$ value in the hypercube (Figure 1b) of the speed space $\Gamma$, and executing $P$ under $\gamma$ produces a trace $\tau$. Various probability distributions can be easily defined over a hypercube, compared with a discrete tree. For example, we can sample the schedules that particular threads run excessively fast/slowly to manifest many extreme-case schedules. Sampling of $\Gamma$ is later discussed in Section 3.4.

If we considered all execution traces $\tau \in \Sigma$, multiple traces $\tau_1, \tau_2$ may be represented by the same speed vector $\gamma$, as long as the threads run in a similar relative speed. Furthermore, there may be $\tau \in \Sigma$ which no speed vector can produce. This is intentional because we would like to constrain threads’ execution speeds at a coarse-grained level (each thread is controlled by only a numeric value).

#### 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 $y_i : y_j$. 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**: $S_{nfs}$ attempts to guarantee the fairness of threads where each thread has the equal time interval to execute. So the speed vector $y_{nfs}$ for $n$ threads assigned by $S_{nfs}$ is $[\frac{1}{n}, \ldots, \frac{1}{n}]$ which implies that all threads can be scheduled to execute the same number of events in any interval.

- **The PCT scheduler for concurrency bug manifestation**: $S_{pct}$ 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, $S_{pct}$ sets the speed vector $[\epsilon, \ldots, \epsilon, y_i = 1, \ldots, \epsilon]$ initially, and then changes it to $[\epsilon, \ldots, y_i = 1, \ldots, \epsilon]$.

#### 3.3.3 Concurrency Bugs in the Speed Space

The most important property of the speed space is that it preserves many bug patterns, and thereby sampling in it is capable of finding 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 reversed. 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 $e$ 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 $\pi_e$ is the number of events performed in $\tau_{e, t_i}$ 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 is shown in Figure 3, which
Algorithm 1: Systematic Speed Control

Input: Program $P$ and threads $T$

Var: $\gamma$ - speed vector
Var: $V = \{2^{-k}, \ldots, 2^{-1}\}$ - speeds to assign

1. for $(t_i, t_j) \in T \times T$ and $i \neq j$ do
2. for $(v_1, v_2) \in \{2^{-(k+1)}, 1\} \times V$ do
3. $\gamma \leftarrow$ random speed vector
4. $(\gamma_1, \gamma_2) \leftarrow (v_1, v_2)$
5. Execute-with-Speed($\gamma$)

Figure 4: Sampled speed vectors of a basis $(t_1, t_2)$ in the speed space $\Gamma$.

Algorithm 2: Execute threads with Speed

Input: $T$ - set of total threads
Input: $\gamma$ - speed vector
Var: $W$ - waiting list
Var: $L$ - number of total events to execute in an interval

1. Procedure Execute-with-Speed($\gamma$)
2. for each interval do
3. parallel-for $t \in T$ do
4. Schedule($t, \gamma_1, 0$)
5. $W \leftarrow \emptyset$
6. $\triangleright$ signal threads in $W$
7. $\triangleright$ wait for signal
8. if $d = \gamma_1$ then
9. $W \leftarrow W \cup \{t\}$
10. $\triangleright$ enforce event
11. Schedule($t, s, d + 1$)

4.4 Systematic Sampling of The Speed Space

We propose an algorithm for sampling the interleaving space, which is inspired by the idea of exponential backoff in computer networking: a thread waits for $2^{0}, 2^{1}, \ldots, 2^{n}$ units of times to obtain the exclusive access to a contented resource. Similarly, we systematically enumerate all thread pairs $(t_i, t_j)$ and systematically try all possible "backoffs" between $t_i$ and $t_j$, as described in Algorithm 1.

For a single basis $(t_1, t_2)$, we systematically control the speed assignments using the following speed vectors:

1. $\gamma_1 = 2^{-(k+1)}$ and $\gamma_2 \in V = \{2^{-k}, \ldots, 2^{-1}\}$;
2. $\gamma_1 = 1$ and $\gamma_2 \in V = \{2^{-k}, \ldots, 2^{-1}\}$.

The speeds of other threads are set uniformly random in $(0, 1]$. In other words, the speed value $\gamma_1$ is set to either maximum (1) or minimum (2$^{-(k+1)}$), while $\gamma_2$ enumerates all speed values of 2$^{-k}$.

The sampled points can be reflected in the hypercube of $\Gamma = (0, 1]^n$. Fixing the basis $(t_1, t_2)$ yields a sub-cube of dimension $n - 2$ which is uniformly random sampled. Consider a multithreaded program of $T = \{t_1, t_2, t_3\}$ and $(t_1, t_2)$ and $(t_2, t_3)$ are set as the basis. Sampled points are shown in Figure 4. For $\gamma = [\gamma_1, \gamma_2, \gamma_3] \in \Gamma$, points $(\gamma_1, \gamma_2)$ are always on the boundary of $\Gamma$ projected to the $t_1-t_2$ plane, and $\gamma_3$ is sampled uniformly random (illustrated by the blue segment in the figure). In the more general case that the basis is not fixed, all six surfaces of the cube $\Gamma$ 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 missed in runs under a native scheduler. $S_{nf}$ is just a single point in the diagonal line\(^1\). 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 $\gamma_1 : \gamma_2 = 1$ ($2^0$) because it is the default behavior of $S_{nf}$.

4 SCHNAUZER

In this section, we describe our prototype tool–Schnauzer which cascades the speed control scheduler and trace-based analysis to test multithreaded programs effectively.

Schnauzer consists of three components: a systematical sampling module, a speed control scheduler for executing $P$ with a speed vector $\gamma \in \Gamma$ to obtain an execution trace $r$, and a trace-based analyzer for predicting bugs/anomalies using $r$.

4.1 The Speed-Controlled Scheduler

To yield schedules from speed vectors, we need an scheduler that can control the threads in enforcing events. We describe how an execution trace is generated according to a speed vector in this section.

Algorithm 2 presents the scheduler $S_{Sce}$ (Speed Control Scheduler) which controls the threads with speed vectors. Procedure Execute-with-Speed is called with speed vector $\gamma$ to schedule the threads. It divides the execution into intervals with predefined

\(^1\)All speed vectors in the form of $\alpha \cdot [1, 1, \ldots, 1]$ are equivalent.
length \( L \), which is the number of total events to execute in an interval. During each interval, all threads are governed to enforce events concurrently by the procedure Schedule. When a thread executes enough amount of events, it’s added into the waiting list \( W \) and waits for a new interval to run.

Procedure Schedule is responsible to execute events with speeds for a single thread. It accepts thread identifier \( t \), its corresponding speed assignment \( \gamma_t \) and the number of events \( n_t \) has already enforced. Before enforcing events, we judge whether \( t \) has run up to the amount of events, i.e., \( \frac{n_t}{L} \) equals \( \gamma_t \) at line 2. If \( t \) has enforced enough events, then it should be added into the waiting list and can no longer execute in this interval. Otherwise, \( t \) enforces an event and continue to run. Note that some events may lead thread \( t \) to be blocked. For example, when \( t \) is to acquire a lock \( l \) held by thread \( u \) which is in the waiting list at present, \( t \) must be blocked according to the semantics of locks. If \( t \) is scheduled to execute after \( u \) has been blocked, it just enforce an nop event. Because no threads are assigned with zero speed by our definition, there must be one thread enforcing an meaningful event (not nop operation), i.e., the process does not stall.

4.2 Trace-based Analysis

Trace-based analyses are usually applied to detect concurrency faults of multithreaded programs when given an execution trace. One of the fundamental techniques is HB-based analysis which is on top of the happens-before relation \( \prec_{hb} \) [26]. The happens-before relation is a partial order over events such that if event \( a \) is executed before \( b \) by the same thread then \( a \prec_{hb} b \) or if \( a \) and \( b \) lock/unlock the same lock or create/join a thread then \( a \prec_{hb} b \). Relation \( \prec_{hb} \) is transitive. Two events form a HB data race when they access the same memory location while are not ordered by \( \prec_{hb} \) and at least one of them is a write event.

HB-based approaches [4, 13, 27] are often simple but limited due to their overly conservative HB relation construction which forms extra happens-before relations. There are more powerful techniques proposed recently. Causally-precedes (CP) [21, 40] 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) [17, 18] soundly reorders events in an execution trace by SMT (satisfiability modulo theories) solvers [9]. It can predicts concurrency faults unseen in the input trace.

Though both CP and PTA are capable of exposing bugs in exercised schedules the results of all these analyses heavily depend on the input execution trace [21]. We found that our speed control scheduler which excels in exercising unusual schedules is orthogonal to the trace-based analysis, so we can cascade these two techniques such that our schedule sampling technique can supply better schedules for trace-based analyses to detect concurrency faults.

4.3 Implementation

The speed control scheduler controls execution speeds of threads at a fine-grained level in Section 3. It controls the number of events every thread can enforce during intervals, whereas our implementation restricts the number of function calls each thread can invoke in any interval. In real systems, events leading to many concurrency bugs are coarsely interleaved [20], which indicates that even though we schedule threads at a coarse-grained level, concurrency bugs can still be manifested. In this way, we reduce the space to sample again based on the speed space.

We also found that a master-worker pattern is widely used in real-world programs where a main thread usually creates worker threads to do the same tasks. This observation implies that there is no need to enumerate all thread pairs, explicated in algorithm 1. We can just select representative threads which doing different tasks to test.

We implement Schnauzer on top of LLVM [2]. To record memory access and synchronizations in runtime, our tool instruments all read/write instructions and pthread function calls. Because we control the threads at function-call level, we should instrument other call instruction as well. Our implementation creates an extra thread which periodically monitoring the states of all threads to prevent deadlocks.

Happens-before based detection approach is chosen as the trace analyser cascaded to the scheduler because previous work has exhibited that if provided with desirable execution traces, it demonstrated considerable detection power [21]. Besides, HB detector is simple to implement and run faster as well.

5 EVALUATION

This section describes the evaluation of our SCS scheduler. The evaluation focuses on answering the following 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 the indirect approach that adopts bug-related quantitative metrics to evaluate the techniques.

The first question is answered by comparing SCS with PCT using a quantitative indicator of the testing thoroughness: the number of statement pairs \( s_1, s_2 \) that are unordered by happens-before \( (\prec_{hb}) \). Both techniques are evaluated against a set of popular open-source projects that are mature and widely deployed, which have also been extensively tested and studied in the previous research [31, 39, 41, 42, 44].

The second question is answered by applying Schnauzer 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 Setups

**5.1.1 Subjects.** The multithreaded programs used in the evaluation are shown in Table 1. We have evaluated SCS scheduler on a set of real-world multithreaded programs of varying complexity used by previous work [41]. All subjects contain a concurrency bug (and patches for fixing them).

**5.1.2 Methodology.** We use the number of happens-before race statements in the execution traces to reflect the degree of schedule diversity. A pair of program statements \( (s_1, s_2) \) is a happens-before race in \( \tau \) if and only if there exists events \( e_1, e_2 \) such that:
Table 1: Real-world multithreaded programs with bugs for evaluation.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LOC</th>
<th>#Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aget</td>
<td>file downloading</td>
<td>2k</td>
<td>4</td>
</tr>
<tr>
<td>Cherokee</td>
<td>Web server</td>
<td>80k</td>
<td>4</td>
</tr>
<tr>
<td>Httpd</td>
<td>Web server</td>
<td>290k</td>
<td>4</td>
</tr>
<tr>
<td>Mysql</td>
<td>relational database</td>
<td>390k</td>
<td>15</td>
</tr>
<tr>
<td>Transmission</td>
<td>bit-torrent client</td>
<td>87k</td>
<td>4</td>
</tr>
<tr>
<td>Pbzip2</td>
<td>parallel compression</td>
<td>2k</td>
<td>3</td>
</tr>
</tbody>
</table>

(1) $e_1$ and $e_2$ are the consequences of executing $s_1$ and $s_2$, respectively.

(2) $(e_1, e_2)$ is a happens-before race, i.e., they are not ordered by $\prec_{hb}$ and at least one is a write.

Races are the root cause of many concurrency bugs [25]. However, even if two statements may race, there can be many synchronization operations to make them ordered by $\prec_{hb}$. In other words, only if statements $s_1$ and $s_2$ are scheduled closer enough, a race can be detected by a happens-before race detector [13]. Since all interleaving space sampling techniques produce a set of execution traces (and also schedules), we naturally use the number of unique racing statement pairs $(s_1, s_2)$ to quantify the diversity of schedules.

We compare 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 represents the most ordinary use case. We used apache ab [33] to send requests to Httpd and Cherokee; Aget and Transmission downloaded an online file of 1GB size; Pbzip2 decompressed a file with 1IM; Mysql executed a script that drives two threads simultaneously insert data into a table.

5.1.3 Experimental Setups. We set the parameters of Schnauzer based on the observation that in all these subjects, there are at most three roles of threads. For example, even though Mysql starts 15 threads in total, about half of them are created to initialize the program and the main thread creates a worker thread each time accepting a SQL request.

Therefore, Schnauzer only considers basis in a fixed set of three threads $T_s \subseteq T$ ($|T_s| = 3$) of different roles in sampling the speed space $^4$, and samples all pairs of $(t_i, t_j) \in T_s$ as basis. We set the parameter $k = 7$ such that the relative speeds range from 1/2 to 1/128. There are 42 unique speed vectors (speed vectors of basis $(t_i, t_j)$ and $(t_i, t_j)$ have some overlapping) in total being sampled for each subject. For PCT, we set bug depth $d = 3$ as recommended in the authors’ original paper [5], and sample the same amount of schedules (42) for a fair comparison.

In answering RQ1, we first run Schnauzer once (samples 42 schedules) to see if all concurrency bugs can be manifested using the buggy versions. Then for the quantitative evaluation, we fix these bugs with patches to prevent them from crashing.

In answering RQ2, we use the latest versions of the programs in which the bugs were fixed to find previously unknown ones.

Table 2: Happens-before races detected by SCS and PCT. A pair of statements $(s_1, s_2)$ is a race if the events produced by $s_1$ and $s_2$ are not ordered by $\prec_{hb}$ in at least one testing run.

<table>
<thead>
<tr>
<th>Program</th>
<th>#Data Races</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aget</td>
<td>4</td>
<td>0 (+0%)</td>
</tr>
<tr>
<td>Cherokee</td>
<td>191</td>
<td>121 ( +172%)</td>
</tr>
<tr>
<td>Httpd</td>
<td>1272</td>
<td>-44 (-3%)</td>
</tr>
<tr>
<td>Mysql</td>
<td>60</td>
<td>14 (+30%)</td>
</tr>
<tr>
<td>Transmission</td>
<td>71</td>
<td>16 (+29%)</td>
</tr>
<tr>
<td>Pbzip2</td>
<td>24</td>
<td>2 (+9%)</td>
</tr>
</tbody>
</table>

All experiments were conducted on a desktop with 3.40GHz Intel Core i7 and 16GB memory running Ubuntu 16.04. In answering RQ1, both PCT and SCS were executed for 10 independent testing runs and statistical data is collected. In answering RQ2, the latest version of the programs were run exactly once.

5.2 Experimental Results

Table 3 summarizes the overall experimental results. The number of unique racing statement pairs $(s_1, s_2)$ of SCS and PCT are displayed in Columns 2 and 3, respectively.

Summing values in the table, SCS found 109 (7.2%) more pairs of unique racing statements compared with PCT. Excluding Aget which adopts a trivial concurrency pattern, SCS and PCT found different set of racing statement pairs. SCS outperforms PCT for Cherokee (+172%), Mysql (+30%), Transmission (+29%), and Pbzip2 (+9%) while found 44 (-3%) less pairs than PCT.

5.2.1 RQ1. First, all known concurrency bugs can be successfully manifested by Schnauzer: assertions are violated in running all subjects.

For the quantitative study, the detailed statistics of racing statement pairs $(s_1, s_2)$ of SCS and PCT are displayed in Figure 5. In each figure, the x axis denotes the number of run and the y axis denotes the number of unique pairs of racing statements (RS). The left sub-figure of each subject displays the number of RSes in a randomly chosen testing run of PCT and SCS (a point $(x, y)$ denotes that the $x$-th trace contains $y$ RSes). The right sub-figure of each subject displays the distribution of cumulated RSes (a point $(x, y)$ denotes that there are $y$ RSes in all previous $x$ traces). Shaded area is the 25%-75% percentile.

The subfigures on the right for each program show that SCS can detect more unique data races for programs as the growth of sampled schedules excepting Aget and Httpd. The former is a small-scale application for which both SCS and PCT is able to detect all distinct races (detected in our experiment) in each run and can not detect more races. On the latter program, though SCS detected less data races, the third quantile indicates that SCS is capable of detecting almost the same data races as PCT in some testing rounds. On remaining benchmarks, SCS has a better trend in detecting data races with the increment of sampled schedules, especially on benchmarks Cherokee, Mysql and Transmission. However, PCT tends to detect the same set of data races on these programs, i.e., it

---

4 For example, Schnauzer samples the master thread and two worker threads in testing Mysql.

5 The number counts the union of all racing statement pairs in all ten runs.
does not find new races when sampling schedules. In other words, it exercises the schedules containing the same races repeatedly.

Figure 6 shows a typical schedule from Pbzip2 sampled by SCS but not by PCT. If we attempt to sample this schedule by PCT, we should first set $t_2$ with a higher priority; otherwise, $t_1$ will always spin on variable x. After $t_1$ writes x, it must be changed to a lower priority, thus, $t_1$ is able to exit the while loop and continue to execute in the critical region. 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 in 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, then, $t_1$ may enter into the critical region earlier than $t_2$ because it runs faster.

In summary, SCS outperformed PCT in sampling schedules on most benchmarks. Though it sampled less diverse schedules on Httpd, the results also show that it has the potential of exploring the same number of diverse schedules as PCT on that program.

5.2.2 RQ2. To our surprise, applying Schnauzer to the latest versions of the subjects found previously unknown concurrency bugs. Schnauzer directly crashes the Transmission Client, and found a rarely occurring race in Cherokee Web server. In total, we found three previously unknown bugs, and all bugs were buried for a long time before Schnauzer revealed them. For example, the bug in Cherokee exists since the 1.0 version before the project is migrated to Github. We believe that other concurrency testing work also tested Cherokee under various settings, however, we were the first to find the bug.
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 \(<_{hb}\) may not be a true race: the soundness of a \(<_{hb}\) race detector only held on the first race detected, as for all subsequent racing events may not even happen. For instance, in Figure 6, \text{Read}(t_1, x)\) and \text{Write}(t_2, x)\) are detected as a race because of the ad hoc spinning.

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

1. \(e_1\) or \(e_2\) is not manifested by PCT; or
2. \(e_1\) and \(e_2\) are always ordered by \(<_{hb}\) in the traces sampled by PCT.

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

Model Checking: The goal of model checking [12] is to enumerate all possible schedules systematically, in theory it is able to find all potential concurrency bugs but the enumeration is infeasible because of the schedule explosion problem. Some approaches [7, 16, 43] have been proposed to reduce the interleaving space, while model checking for large programs is still challenging. There are good reasons for enforcing a subset of schedules, e.g., enumerating the schedules in a bounded interleaving space such as delay-bounding [11] and preemption-bounding [28]. Model checking in a bounded space is feasible though time-consuming, but the bounded space also limits the schedule exploration ability, i.e., only buggy schedules in the bounded space can be exercised.

Random Sampling: Other techniques to enforce a schedule subset are sampling in an interleaving space. Random sampling techniques usually insert random sleeps, execution delays, or thread suspensions at synchronization points [35] or choose priority changing points stochastically [5] during program executions. These techniques are simple and able to exercise a large number of schedules in limited time. They may even provide us with a probability guarantee on detecting concurrency bugs. However, complex bugs can frustrate random sampling because it is prone to sample schedules with the same results in the interleaving space and schedules with specific event orderings are hard to exercise. Besides, existing random sampling approaches work in a discrete interleaving space, and so it is difficult to sample uniformly because it is infeasible to know the distribution of schedules. Though our speed control approach assigns random speeds to some threads, we can sample uniformly in the speed space. Furthermore, we exercise particular speed assignments systematically in the speed space that enforce adversary/abnormal schedules.

6 RELATED WORK

Much prior work has focused on exercising buggy schedules effectively for a multithreaded program. These techniques can be broadly classified into three categories: model checking, random sampling, and heuristic sampling.

Figure 6: A typical schedule sampled by SCS but not by PCT
predict concurrency bugs unseen in collected schedules; however they are sensitive to the input schedules. Our scheduler can exercise more adversary/abnormal schedules and help trace-based analysis detect more subtle bugs.

7 CONCLUSION
In this paper, we introduce the concept—speed space and propose an approach to projecting a multithreaded program’s interleaving space to it by assigning threads with various execution speeds. We also present an algorithm to sample schedules uniformly in the speed space, which aims at adversary/abnormal schedules. Trace-based analysis techniques can be cascaded with the speed control scheduler and this combination helps test multithreaded programs effectively. Our implementation—Schnauzer is promising in testing multithreaded programs that it detected unknown concurrency bugs for real-world programs. In future, we will extend our basic model of the speed space proposed in this paper to further validate its ability on sampling schedules of multithreaded programs.

8 ACKNOWLEDGMENTS

REFERENCES

Figure 7: A previously unknown bug in the Cherokee Web server

```c
1 thread->exit = true;
2
3
4 cherokee_thread_wait_end(thread) {
5    if (thd->ended) return ret_ok;
6 }
7
8 free(thd);
9 ...
```


Testing Multithreaded Programs via Thread Speed Control ESEC/FSE 2018, 4–9 November, 2018, Lake Buena Vista, Florida, United States