Front.Comput.Sci.
DOI

On Interleaving Space Exploration of Multi-threaded Programs

Dongjie CHEN[1,2], **Yanyan JIANG** (✉)[1,2], **Chang XU** (✉)[1,2], **Xiaoxing MA**[1,2]

1 State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
2 Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China

Abstract Exploring the interleaving space of a multi-threaded program to efficiently detect concurrency bugs is important but also difficult because of the astronomically many thread schedules. This paper presents a novel framework to decompose a thread schedule generator that explores the interleaving space into the composition of a basic generator and its extension under the "small interleaving hypothesis". Under this framework, we in-depth analyzed research work on interleaving space exploration, illustrated how to design an effective schedule generator, and shed light on future research opportunities.

Keywords Survey, Testing, Concurrency Bugs, Interleaving Space

1 INTRODUCTION

Concurrency bugs are notoriously hard to find that have resulted in serious consequences as the prevalence of multi-threaded programs. The research community decades worked on static and dynamic program analysis approaches to the validation and verification of multi-threaded programs. Static analysis works by summarizing source code properties without actually executing it [1–4]. This paper focuses on dynamic analysis which works by checking against specification violations on an observed multi-threaded program's execution.

Dynamic analysis parses *a single observed execution trace* under a particular thread scheduling. Multi-threaded programs are non-deterministic where different thread schedules may lead to different results even under a fixed input. Concurrency bugs are usually hidden in specific thread interleavings. Based on the execution trace, dynamic analysis can precisely identify a wide variety of concurrency bugs including races [5–23], atomicity violations [24–31], and deadlocks [32–34].

To conduct dynamic analysis on a multi-threaded program for concurrency bug detection, one must execute the program under diverse thread schedules. A program consisting of n threads and totally k execution steps has $O(n^k)$ distinct thread n^k schedules; however, only few of them manifest concurrency bugs [35]. Various approaches have been proposed to explore the interleaving space from enumeration, random sampling to adopting heuristic guidelines.

As a complement of existing surveys [36, 37], this paper tries to in-depth analyze the state-of-the-art of interleaving space exploration problem in a novel perspective:

First, we treat each interleaving space exploration technique as a *schedule generator* for exercising diverse thread schedules. We provide a formal definition of a schedule generator based on a simplified thread model.

Second, we develop a framework for in-depth studying of existing mechanisms and policies for thread schedule generation. With *small interleaving hypothesis* and the inspiration from prior work, which systematically enumerating schedules with a deterministic scheduler and a delay explorer [38], the framework decomposes a schedule generator into two main parts: a *basic scheduler* and an *extension*. The rationale of such a decomposition is that: the basic scheduler is simple and close to the behaviors exposed by common schedulers, which may not manifest concurrency bugs; while the scheduling extension is used to add extra controls to the basic schedulers, which targets for buggy schedules.

Third, with the framework of schedule generator and its

Received Dec. 14, 2019; accepted Apr. 08, 2020

E-mail: *dongjie_cdj@163.com*, *{jyy,changxu,xxm}@nju.edu.cn*.

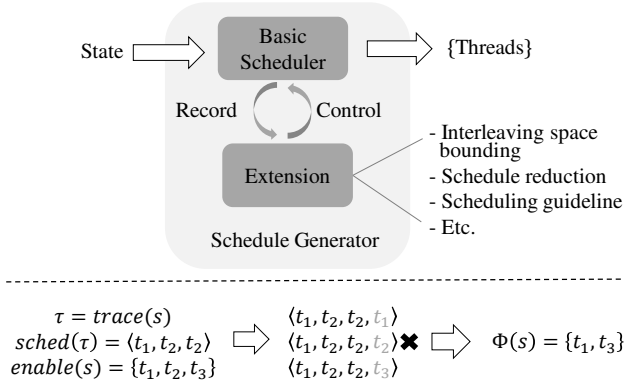

Fig. 1: The Design of Scheduler Generators

decomposition, we survey recent work on how to design a schedule generator. We study different basic schedulers and classify different extensions by their purposes, as shown in Figure 1. Conceptually, the schedule generator first leverages a basic scheduler to explore the common schedules. The basic schedulers define a family of schedules it can expose, and these schedules can be recorded and analyzed to generate new scheduling decisions by extensions. Then, more extra controls are added to the basic schedulers to exercise other different schedules [1)]. For an effective schedule generator, we desire a high-quality basic scheduler that is simple but able to manifest diverse schedules and a delicate extension that adds little additional interference. Our study sheds light on how to construct such a schedule generator by choosing a proper basic scheduler with appropriate extensions.

This paper is laid out as follows. Section 2 makes definitions and gives an overview of our framework on scheduler designs. Section 3 and Section 4 survey recent work to introduce the basic schedulers and to elaborate on different scheduling extensions. Section 5 proposes research opportunities, Section 6 describes related work, and Section 7 concludes this paper.

2 SCHEDULE GENERATORS FOR INTERLEAVING SPACE EXPLORATION

Exploring the interleaving space of multi-threaded programs is crucial to dynamic analysis of multi-threaded programs and has been studied by research communities for a long time [36, 39–41]. After surveying related techniques, we found that most of them can be described as a *schedule generator*, which can be further decomposed for an easier exploration design. In the following sections, we introduce the terminology and their definitions, and then, we present the dynamic analysis procedure and describe the schedule generator formally.

2.1 Notations and Definitions

A multi-threaded program P consists of a set of *threads* $T = \{t_1, \ldots, t_n\}$. Each thread executes 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:

- $\texttt{read}(t, x)/\texttt{write}(t, x)$ for thread t accessing the shared memory variable x;
- $\texttt{lock}(t, \ell)/\texttt{unlock}(t, \ell)$ for thread t performing a synchronization operation on the lock variable ℓ;
- $\texttt{fork}(t, t')/\texttt{join}(t, t')$ for thread t forking/joining a thread t'.

We use $e.\texttt{op} \in \{\texttt{read}, \texttt{write}, \texttt{lock}, \texttt{unlock}, \texttt{fork}, \texttt{join}\}$, $e.t \in T$, and $e.x$ to denote the operation type, the thread, and the variable associated with an event e.

The program execution is modeled as a transition system (S, Δ, s_0), where S is the set of states, $\Delta \in S \times S$ is the *transition* set, and $s_0 \in S$ is the initial state. A state consists of all shared variables and their corresponding values, and a transition $s \xrightarrow{e} s'$ means a new state s' is reached by enforcing an event e in state s. Without ambiguities, we also use $s \xrightarrow{t} s'$ to denote the same transition because event e (enforced by t) is determined by thread t in state s. In each state, threads may be blocked (e.g., acquiring a lock that has been owned by another thread), or active, which are ready to be selected to enforce an event. We denote the active threads as $\texttt{enable}(s)$. An *execution* consists of continuous transitions from the initial state s_0 to the final state s_{n+1}, i.e., $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \cdots \xrightarrow{e_n} s_{n+1}$, where $\forall e_i, e_i.t \in \texttt{enable}(s_i)$.

Chronologically concatenating all events in a program execution yields an *execution trace* τ, a sequence of events $\tau = \langle e_0, \ldots, e_n \rangle$. 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 subsequence of τ. The *schedule* of a trace τ, $\texttt{sched}(\tau)$, is defined as the sequence of thread identifiers in τ: $\texttt{sched}(\tau) = \langle e_1.t, e_2.t, \ldots, e_n.t \rangle$. Given a state s, we can get the trace that leading to s by $\texttt{trace}(s)$. And $\texttt{last}(s)$ represent the last thread in the schedule. Note that in different memory models [42, 43], the same trace may lead to different states, and different traces may lead to the same state. Since we focus on how to explore thread interleaving space, we assume the

[1)] Note that, in practice, the extra controls can work simultaneously with the basic schedulers instead of the pipeline workflow.

Algorithm 1: Schedule Generation

```
   Input: Multi-threaded program P
1  Procedure Dynamic_Analysis()
2      Q ← {s_0}
3      while Q ≠ ∅ do
4          s ← Q \ {s}
5          // Analyze trace(s)
6          for t ∈ Φ(s) do
7              s -t-> s' // State transition
8              Q ← Q ∪ {s'}
```

sequential consistency memory model in this paper as priori work [44–47], i.e., when given fixed inputs, executing the same event sequence always leads to the same result[2].

The schedule generator Φ is responsible to select threads executing events in state s, which is formally defined as $\Phi : S \mapsto 2^T$. In contrast to regular schedulers, e.g., schedulers in operating systems, which always select just one thread each time, the schedule generator may select multiple threads to execute at once in state s, and each selected thread enforces its corresponding event to reach a new state from s.

2.2 Understanding Schedule Generators: A Framework

Dynamic analysis of a multi-threaded program is to execute the program and analyze the execution traces, as shown in Algorithm 1. From the initial state s_0, the dynamic analysis continuously uses the schedule generator Φ to generate new states, which is added to the work-list and exercised later, at Line 6–8. This procedure is similar to stateless model checking [44] except that it relies on the schedule generator to explore the interleaving space instead of enumerating all schedules directly[3]. Thus, the schedule generator is important, which decides which schedules to be checked. Note that the trace can also be logged and analyzed offline.

As the *small scope hypothesis* states: "a high proportion of bugs can be found by testing the program for all test inputs within some small scope" [49]. We believe that for multi-threaded programs, many concurrency bugs can be detected with the "small interleaving", which is characterized by the following hypothesis:

[2] It is also another important problem about how to explore memory model related behaviors.

[3] The dynamic analysis includes model checking and testing, and we name *testing* scheduler here because model checking can be regarded as "super testing" [48].

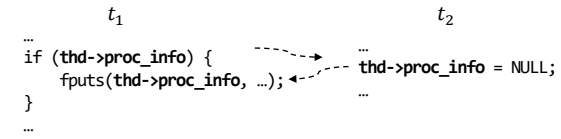

Fig. 2: A Real Concurrency Bug

> SMALL INTERLEAVING HYPOTHESIS. A high proportion of concurrency bugs can manifest by small thread interleaving.

The small interleaving not only means the number of thread switches, e.g., existing work has shown that concurrency bugs can be found by several thread switches [47, 50], but also that it is sufficient to add a little more extra controls to a default scheduler.

Inspired by the small interleaving hypothesis, we decompose a schedule generator into two parts: a basic schedule generator and an extension schedule generator[4]. The former represents a simple scheduler that can also be used directly as a schedule generator, but it may be difficult to manifest concurrency bugs; while the latter is used to add extra controls to disturb schedules from the basic.

We proposed the framework of a schedule generator as:

$$\Phi = \Phi_{basic} \times \Phi_{extension}.$$

This framework also coincides with the bug manifestation. Consider a bug presented in existing work [50], as shown in Figure 2, to manifest it, both threads should execute under a default OS scheduler for long to reach the code snippet; then, small interleaving is controlled to trigger the bug. We can see that: first, the default scheduler is important because most thread executions depend on it; second, its quality largely affects bug finding capabilities because it controls threads to reach the potential buggy code snippet; third, it is, however, insufficient to trigger bugs directly because events from t_1 usually enforced continuously, which means that we need extra controls.

We also generalize the framework further, where the extension generator not only adds extra controls to disturb basic schedulers, but also may generate new schedules, explore more relaxed behaviors, as shown in Figure 1. For example, the schedule obtained from the current state s is $\langle t_1, t_2, t_2 \rangle$, and the enable set is $\texttt{enable}(s) = \{t_1, t_2, t_3\}$. A basic scheduler may let the current thread t_2 continue to execute or preempt

[4] For brevity, we name them "basic scheduler" and "extension", separately, and use these terminologies interchangeably without ambiguities.

it with t_1 or t_3, which results in three possible new schedules. However, the extension can add extra controls by pruning the second schedule, and thus, generating a thread set $\{t_1, t_3\}$ as candidates for state s.

2.3 Discussion

"If the small scope hypothesis holds, it is more effective to do systematic testing within a small scope than to generate fewer test inputs of a larger scope" [49]. Analogously, it is also more effective in systematic testing within a small interleaving space when our hypothesis holds. Existing work can support this to extent[5)]: model checking in a bounded interleaving space is more effective than pure random testing [45, 46, 51]; systematic random testing in a bounded interleaving space is also more effective than pure random testing [47, 52].

Furthermore, many current techniques also follow the small interleaving hypothesis: a multi-threaded program first works as a sequential program; therefore, when generating workload for them, we can generate a sequential prefix with a concurrent short/simple suffix [53]; most concurrency bugs manifest within only two or three threads [50]; we can trigger concurrency bugs by few thread switches [47, 50]; the events involved in a concurrency bug may be close on the timeline [54]. We have tried to design a new schedule generator according to the hypothesis. It consists of a new basic scheduler and extensions, which will be discussed in Section 5.

3 BASIC SCHEDULE GENERATORS

Basic schedulers are simple and common where multi-threaded programs exercise stable schedules, i.e., concurrency bugs may not manifest by these schedules. We introduce some widely-used basic schedulers in the following.

Enumeration Φ_{enum}. Enumeration is the intuitive approach to exercise schedules, which selects all possible threads in state s, i.e.,

$$\Phi_{enum} = \texttt{enable}(s).$$

It is the fundamental of software model checking multi-threaded programs [44]. Since programs execute by repeatedly selecting a thread t to enforce an event e in state s until the execution terminates, and Φ_{enum} enumerates all possible threads in each state, the interleaving space, from the perspective of Φ_{enum}, can be regarded as a tree that is the traditional representation, as shown in Figure 3.

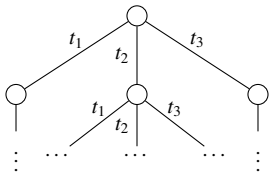

Fig. 3: Interleaving space as a tree. There are three threads, each node represents a state, and the edge is the thread selection.

Φ_{enum}, in theory, is able to detect all concurrency bugs because it enumerates all schedules; however, it is intractable to enforce the enumeration in practice as the number of schedules is astronomical. We usually exploit extra extensions to prune unnecessary schedules when enumerating them, which are introduced in the following section.

Random Φ_{rand}. Random scheduler selects a thread to execute randomly in state s, i.e.,

$$\Phi_{rand} = \{\texttt{rand}(\texttt{enable}(s))\}.$$

It is widely used by stress testing [47] and other techniques [55, 56], but it is usually inefficient because buggy schedules are not uniformly distributed [57], which means that there is a low probability to find buggy schedules because most schedules do not manifest bugs. Thus, many extensions with heuristics or probability guaranteed approaches have been proposed to work with Φ_{rand}.

Round Robin Φ_{rr}. Round robin scheduler selects threads in a predefined order one by one [58]; if the selected thread is blocked, the follow-up thread is chosen. Suppose the order O is defined for each thread and comparable, and the distance between two threads is also computable, It is formalized as:

$$\Phi_{rr} = \{t \,|\, t \in \texttt{enable}(s) \wedge O_{\texttt{last}(s)} \preceq O_t \wedge \texttt{min}(\| O_t - O_{\texttt{last}(s)} \|)\},$$

which means to pick a next available thread with the minimum distance.

Priority Φ_{prio}. Priority-based scheduler assigns each thread with a priority, and it always selects the thread with the highest priority [59] to execute. We use Θ_t to denote the priority of thread t, then,

$$\Phi_{prio} = \{t \,|\, t \in \texttt{enable}(s) \wedge \texttt{max}(\Theta_t)\}.$$

Extra controls can be used to adjust the priority assignments, which will change the schedules [6)].

[5)] Although outcomes do not imply the hypothesis

[6)] Note that Φ_{prio} can be use to simulate other basic schedulers, such as Φ_{rand}, Φ_{rr}, but such a simulation is tricky and needs delicate priority assignments; so there is no need to use Φ_{prio} to substitute other basic schedulers.

Speed Controlled Φ_{scs}. Speed control scheduler [60] assigns each thread with a speed value. All these values form a speed vector Γ, which represents that in an unit-time interval (an epoch) with length L, the number of events enforced by any thread pair $t, t' \in T$ is subject to the ratio of their speed values $\Gamma_t : \Gamma_{t'}$. For a sufficiently large epoch size $L < |\tau|$, as the trace τ becomes longer, we have

$$\lim_{L,|\tau|\to\infty} \frac{\left|(\tau_{k:k+L})\restriction_t\right|}{\left|(\tau_{k:k+L})\restriction_{t'}\right|} = \frac{\Gamma_t}{\Gamma_{t'}}$$

for any $1 \le k < |\tau| - L$; here, $|\tau|$ is the number of events in trace τ.

In practical implementations, each thread has an *execution deposit*, $\Gamma_t * D$, which represents the number of events a thread t can enforce during the current interval, where D is a constant value. Thread t also owns a counter C_t that counts the number of events it has enforced in the current interval. Φ_{scs} selects one thread that still has execution deposits to run. If there are multiple available threads, it selects one of them randomly. Thus, we have

$$\Phi_{scs} = \{\texttt{rand}(\{t \mid t \in \texttt{enable}(s) \ \wedge \ C_t < \Gamma_t * D\})\}.$$

We summarize the basic schedulers in Table 1.

Table 1: Summary of basic schedulers.

Scheduler	Summary
Enumeration Φ_{enum}	Enumerate all possible threads in a state.
Random Φ_{rand}	Randomly select a thread from $\texttt{enable}(s)$.
Round Robin Φ_{rr}	Select threads in an predefined order.
Priority Φ_{prio}	Select the thread with the highest priority.
Speed Controlled Φ_{scs}	Randomly select a thread that still has execution deposits.

4 EXTENSION SCHEDULE GENERATORS

According to our small interleaving hypothesis, extra extensions are responsible to reinforce the detection power of basic schedulers. We present different extensions in this section, which can be used to prune the interleaving space, to provide guidelines for basic schedulers to manifest bugs or to expand execution traces.

4.1 Classifications of Extension Schedule Generators

We classify extension schedule generators into three main categories according to their usages.

Since extensions are used to help basic schedulers, the first main category is to reduce the interleaving space. There are two primary approaches to reducing the space, i.e., bounding the interleaving space directly and avoiding redundant explorations. We name them *interleaving space bounding* and *schedule reduction*, respectively. Particularly, when reducing the space for random exploration, it will have a higher probability to manifest concurrency bugs, thus, named as *probability promotion*. We classify these extensions into a subcategory because these extensions are adapted to random exploration, which is common in concurrency testing.

The second category of extensions is used to explore specific schedules, which contains error-prone multi-threading patterns or expected constraints. The extensions can help basic schedule generators target schedules online with guidelines or offline. The former is classified as *interleaving guideline* further; the latter generates new schedules offline, i.e., *schedule synthesis*.

The last category of extensions attempts to explore more execution behaviors based on basic schedule generators, i.e., exercising *diverse behaviors*.

4.2 Extension for Interleaving Space Bounding

As we present above, the intuitive basic scheduler Φ_{enum} attempts to enumerate all possible schedules, which can work without any extensions because it, in theory, can detect all bugs. However, the interleaving space is astronomically huge in practice, it is necessary to use extensions to prune the interleaving space. A simple approach is to carry out depth first search (DFS) in the interesting space [44] and to bound the depth. As this only covers shallow states (e.g., it may only model check executions of program initializations), it is usually not adopted directly. We introduce other approaches to bounding the interleaving space.

Context bounding is also known as switch bounding. A context is a sequence of events enforced by a single thread, and a thread switch in trace $\tau = \langle e_0, \ldots \rangle$ is defined as

$$\exists i,\ 0 < i < |\tau| \wedge e_i.t \neq e_{i-1}.t.$$

Take trace $\tau = \langle e_1, e_2, e_3, e_4, e_5 \rangle$ as example, and its schedule is $\texttt{sched}(\tau) = \langle t_1, t_2, t_2, t_3, t_1 \rangle$; then, there are four contexts: $\langle e_1 \rangle$, $\langle e_2, e_3 \rangle$, $\langle e_4 \rangle$, and $\langle e_5 \rangle$, which also means there are three thread switches. Thus, the context bounding extension

Table 2: Summary of Scheduling Extensions.

Technique (Year)	Basic× Extension	Summary
Context/Preemption-bounded (2005)	$\frac{(\Phi_{enum}, \Phi_{rand}, \Phi_{prio}, \Phi_{scs}) \times}{\text{Bounding interleaving space}}$	It exercises schedules containing at most k contexts or preemptions [45, 51].
Delay-bounded (2011)	$\frac{(any) \times}{\text{Bounding interleaving space}}$	It transforms the traditional interleaving space into a binary-tree space, and bounds the number of invoking delay explorer [46].
Verisoft (1997)	$\frac{(any) \times}{\text{Schedule reduction}}$	It filters out threads that are independent with the `last`(s) when enumerating new states from state s [44].
DPOR (2005) Inspect (2008)	$\frac{(any) \times}{\text{Schedule reduction}}$	It inserts backtracking points where conflicted events can be executed forward to result in different schedules during explorations [61, 62].
Lapor (2019)	$\frac{(any) \times}{\text{Schedule reduction}}$	It regards lock acquisition events as unordered, identifies critical sections with conflicted read-write events, and just explores different schedules between conflict critical sections [63].
RAPOS (2007)	$\frac{(\Phi_{rand}, \Phi_{scs}) \times}{\text{Higher probability}}$	It randomly selects multiple threads to enforce events concurrently, where each thread's event is independent with each other but is dependent on the last event `last`(s) in state s [35].
PCT (2010)	$\frac{(\Phi_{prio}) \times}{\text{Higher probability}}$	It exploits the priority scheduler and randomly samples d schedule points, at which it lowers the priority of the current thread, thus, switches to another thread [47].
PPCT (2012)	$\frac{(\Phi_{prio}) \times}{\text{Higher probability}}$	It is similar to PCT except that it maintains two thread priority sets, where any thread in the high priority set is allowed to execute [52].
POS (2018)	$\frac{(\Phi_{prio}) \times}{\text{Higher probability}}$	It is based on Φ_{prio}, and attempts to sample a schedule satisfying a partial order by randomly reassigning priorities to each event in the `enable` set after every selection [57].
ConTest (2005)	$\frac{(\Phi_{rand}, \Phi_{rr}, \Phi_{prio}, \Phi_{scs}) \times}{\text{Guideline}}$	It guides the exploration to cover higher synchronization coverages [64].
RaceFuzzer (2008)	$\frac{(\Phi_{rand}, \Phi_{rr}, \Phi_{prio}, \Phi_{scs}) \times}{\text{Guideline}}$	It uses an unsound data race detector to find potential racing statement pairs and guides sampling schedules manifesting these races [55].
Ctrigger (2009)	$\frac{(\Phi_{rand}, \Phi_{rr}, \Phi_{prio}, \Phi_{scs}) \times}{\text{Guideline}}$	It records schedules, analyzes atomicity violation patterns in these traces, and attempts to insert delays to result in the potential bugs [27].
DeadlockFuzzer (2009)	$\frac{(\Phi_{rand}, \Phi_{rr}, \Phi_{prio}, \Phi_{scs}) \times}{\text{Guideline}}$	It uses an unsound deadlock detector to find potential bugs and guides sampling schedules manifesting these deadlocks [56].
HaPset (2011)	$\frac{(\Phi_{rand}, \Phi_{rr}, \Phi_{prio}, \Phi_{scs}) \times}{\text{Guideline}}$	It attempts to cover more statement pairs, where one statement enforcing an event is immediately dependent upon another [65].
Concurrit (2012)	$\frac{(\Phi_{enum}) \times}{\text{Guideline}}$	It adopts a cooperative model checking strategy where it allows programmers to specify interesting schedules by customized annotations in the programs [66].
SyncPair (2012)	$\frac{(\Phi_{rand}, \Phi_{rr}, \Phi_{prio}, \Phi_{scs}) \times}{\text{Guideline}}$	It uses the *synchronization-pair* coverage to guide the exploration [67].
DrFinder (2015)	$\frac{(\Phi_{rand}, \Phi_{rr}, \Phi_{prio}, \Phi_{scs}) \times}{\text{Guideline}}$	It attempts to reverse acquisition-release relations to find hidden data races [15].
CheckMate (2010)	$\frac{(\Phi_{enum}) \times}{\text{Synthesis}}$	It uses customized annotations to abstract traces into trace programs containing only synchronization events, and model checks the generated programs to find deadlocks [68].
PENELOPE (2010)	$\frac{(any) \times}{\text{Synthesis}}$	It abstracts a given trace and use its algorithmic analysis to generate new schedules that contain atomicity violations [69].
Racageddon (2014)	$\frac{(\Phi_{rand}, \Phi_{rr}, \Phi_{prio}, \Phi_{scs}) \times}{\text{Synthesis}}$	It uses the concolic testing to find an input and a schedule manifesting given data races [70].
MCM (2014)	$\frac{(any) \times}{\text{Synthesis}}$	It encodes a given trace concurrency bug patterns into ordering constraints, and uses SMT solvers to generate new schedules [71].
Musuvathi et al. (2008)	$\frac{(\Phi_{enum}) \times}{\text{Diverse behaviors}}$	It enforces fair stateless model checking: it records which threads are blocked by t, and resumes blocked thread when t yields [72].
Memcheck (2011) CDSchecker (2013)	$\frac{(\Phi_{enum}) \times}{\text{Diverse behaviors}}$	It takes into account more low-level events that accept relaxed consistency guarantees, encode memory coherence guarantees offered by the C11 semantics, and model check programs with these extra constraints [73, 74].

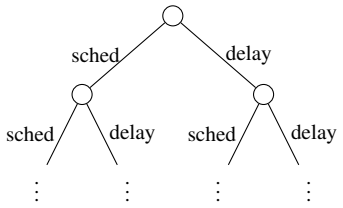

Fig. 4: Search space of delay bounding.

restricts basic schedulers to exercise schedules containing at most k contexts, or $k-1$ switches [51].

Preemption bounding is similar to context bounding except that a thread preemption occurs when the last thread is still active. We define preemption in trace $\tau = \langle e_0, \ldots \rangle$ as:

$$\exists i,\ 0 < i < |\tau| \wedge e_i.t \neq e_{i-1}.t \wedge e_{i-1}.t \in \texttt{enable}(\texttt{state}(\tau_{0:i-1})).$$

Preemption bounding extensions, thus, restricts basic schedulers to explores schedules with at most k preemptions [45, 75].

Delay bounding extension in model checking [38, 46] is ingenious, which combines a deterministic scheduler and a delay explorer. The delay explorer is to block the current thread, i.e., delaying the execution of the current thread. During the interleaving exploration, instead of selecting which thread to execute, it makes a binary decision: let the deterministic scheduler select a thread to execute or let the delay explorer block the current thread and select a next thread again. In this way, it transforms the traditional interleaving space into a binary-tree space, as shown on Figure 4. Furthermore, it bounds the number of delay explorer invoking. Delay bounding can be used naturally by our schedule generator that bounds the number of extra extension usages for basic schedulers.

All these bounding extensions restrict the space to explore, and thus, make enumerations in the space tractable. However, they also restrict concurrency bugs to detect at the same time. They can also be used with other basic schedulers besides Φ_{enum} and take a larger boundary number, according to our framework. For example, Φ_{scs} can work with context bounding, which restricts the number of thread switches in each interval while each interval execution still conforms to the speed vector of Φ_{scs}.

4.3 Extension for Schedule Reduction

Besides bounding the interleaving space, extensions about partial order reduction (POR) [76] also attempts to reduce the interleaving space to explore by exercising only those schedules resulting in different states. They can filter out equivalent schedules that are redundant to exercise, where two schedules are deemed equivalent if one can be obtained from the other by commuting independent event pairs.

Two events are independent if they are not conflicted and there is not a happens-before relation between them. First, events conflict with each other if they access the same memory location, and at least one of them is a write event. Second, the happens-before relation [43], denoted as $\prec_{hb}$, is a partial order over events such that 1). event e_a is executed before e_b by the same thread, then, we have $e_a \prec_{hb} e_b$; or, 2). event e_a and e_b lock/unlock the same lock or create/join a thread then $e_a \prec_{hb} e_b$. Relation $\prec_{hb}$ is transitive. Thus, trace $\langle \texttt{read}(t_1, x), \texttt{write}(t_2, y), \texttt{read}(t_1, x) \rangle$ is equivalent to trace $\langle \texttt{write}(t_2, y), \texttt{read}(t_1, x), \texttt{read}(t_1, x) \rangle$ because $\texttt{write}(t_2, y)$ can be commuted froward.

Verisoft [44] is the pioneer work of model checking the interleaving space of multi-threaded programs. It works on top of the enumeration scheduler Φ_{enum}, and it also proposed an extension to reduce the to exploration. When enumerating new states from state s, it filters out threads that are independent with the $\texttt{last}(s)$. Note that it only considers two continuous independent events instead of events with happens-before relations.

Other techniques, like dynamic partial order reduction (DPOR) [61, 62], identify different schedules with happens-before relations during runtime. When using basic schedulers to select a next thread enforcing an event, DPOR analyzes whether the selected event conflicts with previously executed events in the trace and whether it can be scheduled forward to execute. If so, DPOR inserts a backtracking point where the event can be executed, which results in an alternative schedule that is not "equivalent" to the current one. For example, if trace $\langle \texttt{read}(t_1, x), \texttt{write}(t_2, y), \texttt{read}(t_1, x) \rangle$ is exercised, and $\texttt{write}(t_3, y)$ is selected now; we may analyze that $\texttt{write}(t_3, y)$ and $\texttt{write}(t_2, y)$ are conflicted and concurrent, and thus, $\texttt{write}(t_3, y)$ may be enforced forward. We can insert a backtracking point right after $\texttt{read}(t_1, x)$, i.e., adding t_3 to its `enable`, which attempting to explore a new trace $\langle \texttt{read}(t_1, x), \texttt{write}(t_3, y), \ldots \rangle$.

Recent techniques further improve DPOR to filter out more schedules. Since most POR techniques take lock acquisition events dependent, i.e., a `unlock` event always has a happens-before relation with its following `lock` event, regardless whether events in the critical sections are conflicted, they consider all possible interleaving between these critical sections. Lock-Aware Partial Order Reduction (Lapor) [63], on the other hand, regards lock acquisition events as unordered, identifies critical sections containing conflicted

read-write events, and just explores different schedules between conflicted critical sections.

4.4 Extension for Probability Promotion

Consider that the interleaving space is astronomically large, we usually sample schedules when testing multi-threaded programs. Nevertheless, pure random schedule sampling by Φ_{rand} may be inefficient. Extensions are used to promote schedule sampling techniques by filtering out redundant samplings or raise probabilities of finding buggy bugs.

RAndom Partial Order Sampling (RAPOS) [35] proposed an extension to reinforce Φ_{rand}, which attempts to sample schedules resulting in different states. During each thread selection in state s, it randomly selects multiple threads to enforce events (each thread enforces one event) concurrently, where each thread's event is independent with each other but is dependent on the last event $\texttt{last}(s)$. Note that it is similar to Verisoft that it only considers dependence between two continuous events but not happens-before relations.

Thread sampling can also be probability guaranteed. Consider that most concurrency bugs can manifest by several thread preemptions [50], Probabilistic Concurrency Testing (PCT) [47] uses this heuristic to sample preemption points in the schedule instead of randomly selecting threads. It proposed a *depth* metric of a bug as the minimum number of thread switches to find the bug. It, thus, samples d preemption points for a bug with depth d. PCT exploits the priority scheduler Φ_{prio}, i.e., it lets one thread with the highest priority to execute continuously, and randomly selects d schedule points, at which it lowers the priority of the current thread, thus, switches to another thread. Assume that there are n threads and k events, then, it is with probability $1/nk^d$ to sample a specific schedule—it is $1/n$ to randomly select the first thread to execute and is $1/k^d$ to select d appropriate preemption points. PCT proves a more restrict lower bound with probability $1/nk^{d-1}$ in their work.

PCT can be further improved as for as execution performance because it schedules only one thread at each thread selection. As an improvement, PPCT (Parallel PCT) [52] schedules multiple threads at a time, which is able to run threads concurrently and still has the same probability guarantee as PCT. In contrast to PCT that always allows a thread with the highest priority to execute, PPCT maintains two thread priority sets, namely higher priority set and lower priority set; any thread with higher priority is allowed to execute. Similarly, PPCT still randomly samples d schedule points to change the priorities of threads.

Though PCT provides a fabulous lower bound of probability, it may be inefficient to find a bug with more preemption schedules, i.e., when d becomes large because PCT is unaware of partial orders and samples redundant schedules. As a contrast, Partial Order Sampling (POS) [57] is a partial order aware technique, and it attempts to sample a schedule satisfying a partial order. Suppose we attempt to sample a schedule satisfying the partial order $e_1 \prec_{hb} e_2 \prec_{hb} e_3$, it should guarantee that $\text{prio}(e_1) > \text{prio}(e_2) > \text{prio}(e_3)$ when using a priority scheduler. We can see that the probability to sample such a schedule is lower as the schedule becomes longer because each assignment depends on previous priorities. On the other side, POS ingeniously observe that it is unnecessary to provide the whole priority sequence but is sufficient to assign priorities separately—the assignment, $\text{prio}(e_1) > \text{prio}(e_2)$ and $\text{prio}'(e_2) > \text{prio}'(e_3)$, is also able to sample the partial order. In this way, POS does not take history into consideration when assigning event priorities, and it can randomly reassign priorities to each event in the `enable` set after every thread selection[7)].

4.5 Extension for Interleaving Guideline

Though extensions like PCT or POS provide a probability guarantee for random schedule sampling, it is still difficult to sample buggy schedules when confronted with real-world programs that contain millions of events. Other extensions have also been proposed to use different heuristics directly that guide exercising buggy-prone schedules.

Similar to code coverage that is commonly used as a metric to measure the sufficiency of testing, various concurrency coverage is used to help exercise different schedules [77]. ConTest [64] uses synchronization coverage to test multi-threaded programs. When a thread is to enter a critical section (i.e., acquiring a lock), it will be in a *blocked* state or a *blocking* state. ConTest attempts to let threads exercise different states as far as possible by injecting delays at synchronization points. Meanwhile, Hong et al. [67] proposed a *synchronization-pair* coverage, which means two continuous lock acquisitions on the same lock without an intermediate acquisition. They also schedule programs before each lock acquisition to produce different synchronization pairs by injecting delays.

Pset describes a more general concurrency coverage on top of statements [78]. The pset of a statement st, $pset(st)$, means that for each statement $st' \in pset(st)$, there exists an exercised

7) If there are multiple events with a same priority, select one of them randomly.

schedule, in which an event enforced by statement st is immediately dependent upon an event enforced by st'. HaPset [65] extends Pset by identifying a statement with more information, e.g., *file*, *line*, *thread* and *calling context*. It guides the interleaving exploration for more pset coverages. For example, if there is a pset $st' \in pset(st)$, when an event is enforced by st', we may delay the current thread. If statement st is then to execute, we also block it and select other events to enforce, such that we look for a new statement immediately depending on st'.

In addition to concurrency coverages, extensions can also use bug patterns to guide the interleaving exploration directly. Ctrigger [27] targets for schedules manifesting atomicity violations. An atomicity violation occurs with the access pattern $\langle e_1, e_2, e_3 \rangle$, where all events access the same variable, and e_1 and e_2 are enforced by the same thread, while event e_3 is from a different thread. During the exploration, Ctrigger can record schedules, analyzes patterns in execution traces, and attempts to delay the event e_3, thus, leading to the expected pattern.

The data race is an important indicator of concurrency bugs. A data race happens between two memory accesses when there are no happens-before relations between them, and one of them is a write operation. RaceFuzzer [55] is capable of strengthening basic schedulers to guide schedules sampling by data races, though it was proposed to reinforce Φ_{rand} originally. It first uses an unsound data race detector to find potential racing statement pairs (i.e., events enforced by the statements are raced); then, it executes the program again. When the selected thread is to enforce a racing statement, it will be delayed until its racing pair is also ready to execute. When both the statements can be executed, i.e., there are two threads in the `enable` to enforce them, a data race can be captured. RaceFuzzer, then, randomly select one of them to execute first.

Instead of exploiting a race detector beforehand to compute statement candidates that may manifest concurrency bugs, DrFinder [15] attempts to reverse happens-before relations to find hidden data races. Suppose that there is a schedule $\texttt{read}(t_1, x) \prec_{hb} \texttt{unlock}(t_1, \ell) \prec_{hb} \texttt{lock}(t_2, \ell) \prec_{hb} \texttt{write}(t_2, x)$, if the ordering $\texttt{unlock}(t_1, \ell) \prec_{hb} \texttt{lock}(t_2, \ell)$ is reversed, the `read` and `write` operation may happen concurrently that comprise a data race. DrFinder proposed a *may-trigger relation* that specifies whether a method may trigger a lock acquisition either directly or indirectly. It first profiles executions to compute the relation, and during exploration, it may suspend a thread when acquiring a lock if other concurrent methods may trigger the acquisition on the same locks. In this way, it attempts to reverse the lock acquisitions by allowing anther thread to hold locks first. For the above-mentioned example, if we know thread t_2 will acquire the lock ℓ soon and thread t_1 is to acquire the same lock, we will block t_1 and let t_2 hold the lock first.

On the other hand, DeadlockFuzzer [56] strengthens basic schedulers by the guideline of deadlocks, which is also a common concurrency bug [50]. A deadlock happens when there exists a lock acquisition dependence cycle, e.g., $\langle \texttt{lock}(t_1, \ell), \texttt{lock}(t_2, \ell'), \texttt{lock}(t_1, \ell'), \texttt{lock}(t_2, \ell) \rangle$. Similar to RaceFuzzer, it uses an unsound deadlock detector (which may cause false positives) to find potential deadlock cycles first. During the exploration, it performs delays when encountering lock acquisitions or lock releases, which target to manifest real deadlock cycles.

Most approaches generate guidelines automatically without human efforts. While, if programmers provide guidelines to exercise schedules, it may be more efficient to detect concurrency bugs. For example, if a function is regarded as atomic, then, a programmer can annotate this function to notify the basic scheduler such that events in the function can be enforced continuously. Concurrit [66] adopts such a cooperative strategy where it allows programmers to specify interesting schedules by annotations in the programs, and it systematically exercises these schedules.

4.6 Extension for Schedule Synthesis

Extensions may cooperate with basic schedulers offline instead of guiding the next thread selection during dynamic analysis, i.e., they accept traces produced by basic schedulers and synthesizes new traces directly. These extensions can be regarded as trace analysis techniques (predicted trace analysis) [71, 79, 80] because they do not control threads during runtime, but we surveyed them in our framework because they reinforce basic schedulers as far as exploring the interleaving space.

Racageddon [70] takes potential data race candidates as inputs and synthesizes schedules resulting in real races. For each data race candidate, it uses the concolic testing [81, 82] to find an input and a schedule manifesting the data race. During the synthesis, more different data races may be exposed in the generated schedules, and it swaps these data races as new candidates to synthesize more schedules.

PENELOPE synthesizes schedules manifesting atomicity violations [69]. Given a trace, PENELOPE abstracts it and subjects it to an *algorithmic analysis* to generate new schedules that contain atomicity violations. These atomicity viola-

tions are restricted, i.e., comprising two threads and a single variable according to the marked boundaries. It uses lockset analysis to identify two concurrent events e_1 and e_2, which means that an atomicity violation sequence $\langle e_1, e_2, e_3 \rangle$ is feasible because e_2 can execute concurrently with e_1, thus, occurs between e_1 and e_3. The generated schedule is not sound, which is feasible at the abstract level but may be infeasible in real executions. PENELOPE hence re-executed schedules with test harness to prune false positives.

Predictive trace analysis is a general technique to analyze execution traces [80, 83], and maximal causal model (MCM) has been proposed to synthesize schedules given traces [71, 79, 80]. It encodes traces together with concurrency bug patterns into constraints, and it leverages SMT solvers [84] to generate new orderings that represent new schedules. Each event in the input trace is assigned an order variable O_e, and order variables should satisfy the ordering constraints, i.e., happens-before relation, read-write constraint, etc. For example, if $e_1 \prec_{hb} e_2$, then, we have $O_{e_1} < O_{e_2}$ in the constraint. All constraints are fed into SMT solvers and produce new schedules manifesting concurrency bugs.

CheckMate [68] is used to generate a trace program instead of a new trace. It annotates programs with specific annotations, which are used to help identify markers in the trace. Then, it generates a trace program containing only synchronization events from the trace. After that, the trace program is analyzed again to detect deadlocks. Since there are not irrelevant events like memory accesses in the trace program, the interleaving space is tractable. However, the trace program is not sound, i.e., the schedules of the trace program may be infeasible as far as the original program.

Note that the new schedules generated by these synthesis extensions is constrained to original input traces, which are obtained by basic schedulers. So, basic schedulers and synthesis extensions benefit each other.

4.7 Extension for Diverse Behaviors

Extensions can also be used to strengthen the dynamic analysis or exercise more execution diversities.

Dynamic analysis in Algorithm 1 produces new states by selecting the next threads iteratively; however, it is inefficient for nonterminating program executions, e.g., a thread doing spin loops may be selected to execute repeatedly. Musuvathi et al. proposed the fair stateless model checking based on the observations: *if thread t is scheduled infinitely often, then in infinitely many of those t transitions thread t also yields* [72]. During the exploration, they record all threads blocked by t; when t is also blocked or yield the execution, the blocked threads by t are selected to continue to execute instead of selecting t again.

We assume sequential consistency above for all techniques, which also limits the detection power of basic schedulers in relaxed memory models. Even by model checking programs on relaxed memory models, exercising only schedules is insufficient because not only schedules influence program states but also other relaxed behaviors. For example, a read event in a deterministic schedule may read different values, which depend on the memory model, i.e., in trace $\langle \texttt{write}(t_1, x), \texttt{write}(t_1, x), \texttt{read}(t_{2,x}) \rangle$ the `read` may read from either the first write or the second one. With certain extensions, we can explore more executions allowed by the memory model. Memcheck [73] and CDSchecker [74] exhaustively explores different behaviors allowed in relaxed memory models. It takes into account more low-level events (e.g., atomic variables' read/write in C11) that accept relaxed consistency guarantees, and it encodes memory coherence guarantees offered by the C11 semantics. With such extra constraints, it is able to explore more behaviors for a schedule. In other words, these extensions interpret schedules with other constraints, which allow different execution results.

We summarize all the mentioned-above extensions in Table 2 according to their purposes and publication years. We also list our recommendations for the cooperation between basic schedulers and extensions.

4.8 Discussion

Note that extensions can be combined into usages; for example, one can use a space bounding extension to reduce the interleaving space, use guidelines to target for specific schedules, and use synthesis extensions to generate new schedules. We list each of them as a representative application of different extensions.

5 RESEARCH OPPORTUNITIES

The framework of schedule generator and the small interleaving hypothesis provides several research opportunities from designing basic schedulers, exploiting different extensions to combining them effectively.

First, we can introduce more basic schedulers, which is important and largely affect the detection capability. We can see that basic schedulers execute threads for long to reach the potential bug site; thus, the state it can reach should be diverse, which means basic schedulers should be diverse. For

example, it can be fair like Φ_{rr}, Φ_{rand}; meanwhile, it can also be extremely unfair like Φ_{scs}. Following this design, basic schedulers can expose more diversity besides fairness. For example, it can expose the affinity between threads and variables such that different threads access different memory locations with various access time, overhead, or speed. It can also expose the fairness of thread groups instead of individuals by grouping different threads into a bundle and schedule them together.

Second, most of the existing basic schedulers work on the event granularity, which incurs high overhead when scheduling. However, we can accelerate them by scheduling thread at a more coarse granularity, like functions other than events, according to the coarse-grain hypothesis [85]. In this way, we can also combine different basic schedulers by layered scheduling, i.e., a coarse-grained scheduler and a fine-grained scheduler. For instance, we may leverage Φ_{scs} to schedule threads at function-level; while, during each epoch, we can exploit Φ_{prio} to schedule events.

As far as extensions, we find that bounding the interleaving space and reducing schedules are two important extensions. However, when using them to reinforce Φ_{enum} and Φ_{rand} separately, we need different concrete techniques to fit basic schedulers. Similarly, we may propose corresponding approaches to adopt these extensions to other basic schedulers.

Consider that there still is not an omnipotent schedule generator to testing all multi-threaded programs, we should design different schedule generators for programs. Studying the characteristics of different programs may shed lights on how to combine basic schedulers and extensions effectively.

We introduce our experience in trying to design a schedule generator—Schnauzer [60]. First, the importance of basic schedule generators inspired us to design a different one in contrast to common fair schedulers like Φ_{rand} and Φ_{rr}. The speed control scheduler Φ_{scs} is the outcome, which works as the basic schedule generator of Schnauzer. Then, it controls speeds of few representative threads with different roles instead of all threads according to the small interleaving hypothesis; in addition, it also reduces the interleaving space by scheduling threads at a coarse-grained level. Schnauzer is able to reveal unknown concurrency bugs hidden for a long time, and it can be further improved by combining other extensions. For example, we can bound the number of different speed vectors to exercise; we can bound the thread contexts/preemptions during each execution interval, which means bounding the interleaving sub-space of each interval. We believe that designing new basic schedule generators and applying different extension schedule generators can be future work of developing a more effective schedule generator.

6 RELATED WORK

As a great number of techniques have been proposed to analyze multi-threaded programs, there are also lots of prior work surveying these approaches and building a roadmap of them.

Exploring interleaving space in symbolic execution of multi-threaded programs. We discuss exploring interleaving space in dynamic analysis of multi-threaded programs in this paper; similarly, how to effectively exploring interleaving space is also important to symbolic execution of multi-threaded programs. In contrast to dynamic analysis, symbolic execution of multi-threaded programs takes inputs as symbolic values. For each execution, it records the schedule and path constraints executed by each thread as the execution constraint; then, it can either select a schedule point to build a new schedule or flip a branch in path constraints, which constructs a new execution constraint that can be solved by SMT solvers [86–89].

As far as exploring interleaving space, dynamic analysis and symbolic execution are similar at a high-level view; i.e., selecting which thread when given a (concrete/symbolic) state during the (concrete/symbolic) execution. Thus, basic schedulers, like Φ_{enum} and Φ_{rr}, can also be used in symbolic executions, and extensions may be designed from a similar perspective, e.g., symbolic execution in a bounded space [87] and using guidelines [86]. Furthermore, these exploration approaches may be tailored to symbolic execution because it not only considers interleaving space but also input space for the effective whole exploration.

Survey of concurrency bugs. Lu et al. conducted an influential survey that comprehensively analyzes concurrency bugs [50]. They studied 105 read bugs from 4 large and mature open-source applications: MySQL, Apache, Mozilla, and OpenOffice, and found important implications about bug patterns, bug manifestations, bug fix strategies, etc. Their study reveals that atomicity-violation or order-violation are two main patterns of concurrency bugs, which inspired a great deal of subsequent checking techniques, and that most bug manifestations are simple where bugs can be triggered if certain order between just 2 threads is enforced, which also provided a strong heuristic to bug detection and affected many approaches.

Survey of concurrency testing. Arora et al. surveyed testing multi-threaded programs based on different high-level

testing approaches [40]. They studied 179 research publications over the period of 1983 to 2014, extracted the approaches from these works of literature, and classified them into different categories, i.e., structural testing, model- based testing, mutation testing, formal method, search-based testing, etc.

More specifically, Bianchi et al. surveyed recent techniques in testing multi-threaded programs [36] from the perspective of concurrency instead of high-level testing approaches. They searched literature from 2000 to 2015 systematically, selected 94 papers that witness novel research contributions, and proposed a testing framework that is analogous to the dynamic analysis. They classified papers into different categories: test generation, interleaving selection, and oracle check, according to their framework; for each category, they made further classifications by different techniques approaches (e.g., dynamic or static ones), different heuristics, etc.

Fu et al. also conducted a systematic review of concurrency testing; besides testing, they surveyed the literature on concurrency bug exposing, detection, avoidance, and fixing [39]. For each category, they then studied existing work and introduced concrete techniques each work adopts.

These surveys are comprehensive studies that cover all techniques of testing multi-threaded programs, but they simply present some techniques related to interleaving exploration instead of analyzing the problem deeply, which is the focus of this paper. We concentrate on how to exercise thread schedules, which can be represented as the problem of how to design a schedule generator, and then, we decompose the schedule generator into different parts and survey existing work about them.

Survey of systematically exploring bounded interleaving space. Thomson et al. presented the first independent empirical study on schedule bounding techniques for dynamic analysis of multi-threaded programs [37]. They compared the effectiveness of depth-first search, preemption bounding [45], delay bounding [46], and random scheduling. The results show that "1) delay bounding is superior to preemption bounding; 2) schedule bounding is more effective at finding bugs than unbounded depth-first search; 3) a naive random scheduler is at least as effective as schedule bounding for finding bug". These are important indicators to develop concurrency testing tools; nonetheless, we surveyed more dynamic analysis techniques and proposed a framework to design interleaving space exploration.

7 CONCLUSION

Exploring the interleaving space is crucial to the dynamic analysis of multi-threaded programs. We regarded it as designing a scheduler generator and decomposed it into two main parts: a basic scheduler and an extension. According to our framework, we surveyed common basic schedulers and work on different extensions. We hope that our framework could shed light on how to design an effective schedule generator and researches on basic schedulers and extensions.

Acknowledgements 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 #2017YFB1001801) of China, National Natural Science Foundation (Grants #61932021, #61690204, #61802165) of China. The authors would like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

1. Voung J W, Jhala R, Lerner S. RELAY: Static Race Detection on Millions of Lines of Code. In: Proceedings of Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. 2007, 205 – 214.
2. Naik M, Aiken A, Whaley J. Effective Static Race Detection for Java. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2006, 308 – 319.
3. Blackshear S, Gorogiannis N, O'Hearn P W, Sergey I. RacerD: Compositional Static Race Detection. In: Proceedings of ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications. 2018, 144:1 – 144:28.
4. Gorogiannis N, O'Hearn P W, Sergey I. A True Positives Theorem for a Static Race Detector. In: Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2019, 57:1 – 57:29.
5. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems, 1997, 15(4):391 – 411.
6. Netzer R. Race Condition Detection for Debugging Shared-memory Parallel Programs. 1991.
7. Smaragdakis Y, Evans J, Sadowski C, Yi J, Flanagan C. Sound Predictive Race Detection in Polynomial Time. In: Proceedings of Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2012, 387 – 400.
8. O'Callahan R, Choi J-D. Hybrid Dynamic Data Race Detection. ACM SIGPLAN Notices, 2003, 38(10):167 – 178.
9. Sheng T, Vachharajani N, Eranian S, Hundt R, Chen W, Zheng W. RACEZ: A Lightweight and Non-invasive Race Detection Tool for Production Applications. In: Proceedings of International Conference on Software Engineering. 2011, 401 – 410.

10. Flanagan C, Freund S N. FastTrack: Efficient and Precise Dynamic Race Detection. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2009, 121 – 133.
11. Marino D, Musuvathi M, Narayanasamy S. LiteRace: Effective Sampling for Lightweight Data-race Detection. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2009, 134 – 143.
12. Bond M D, Coons K E, McKinley K S. PACER: Proportional Detection of Data Races. In: ACM SIGPLAN Notices. 2010, 255 – 268.
13. Prvulovic M, Torrellas J. ReEnact: Using Thread-level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In: Proceedings of Annual International Symposium on Computer Architecture. 2003, 110 – 121.
14. Rajagopalan A K, Huang J. RDIT: Race Detection from Incomplete Traces. In: Proceedings of Joint Meeting on Foundations of Software Engineering. 2015, 914 – 917.
15. Cai Y, Cao L. Effective and Precise Dynamic Detection of Hidden Races for Java Programs. In: Proceedings of Joint Meeting on Foundations of Software Engineering. 2015, 450 – 461.
16. Petrov B, Vechev M, Sridharan M, Dolby J. Race Detection for Web Applications. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2012, 251 – 262.
17. Raychev V, Vechev M, Sridharan M. Effective Race Detection for Event-driven Programs. In: Proceedings of ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications. 2013, 151 – 166.
18. Maiya P, Kanade A, Majumdar R. Race Detection for Android Applications. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014, 316 – 325.
19. Yu Y, Rodeheffer T, Chen W. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In: Proceedings of ACM Symposium on Operating Systems Principles. 2005, 221 – 234.
20. Kasikci B, Zamfir C, Candea G. RaceMob: Crowdsourced Data Race Detection. In: Proceedings of ACM Symposium on Operating Systems Principles. 2013, 406 – 422.
21. Choi J-D, Lee K, Loginov A, O'Callahan R, Sarkar V, Sarkar V, Sridharan M. Efficient and Precise Datarace Detection for Multithreaded Object Oriented Programs. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2002, 258 – 269.
22. Narayanasamy S, Wang Z, Tigani J, Edwards A, Calder B. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. ACM SIGPLAN Notices, 2007, 42(6):22 – 31.
23. Kasikci B, Zamfir C, Candea G. Data Races vs. Data Race Bugs: Telling The Difference with Portend. ACM SIGPLAN Notices, 2012, 47(4):185 – 198.
24. Lu S, Tucek J, Qin F, Zhou Y. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems. 2006, 37 – 48.
25. Flanagan C, Freund S N, Yi J. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2008, 293 – 303.
26. Park C-S, Sen K. Randomized Active Atomicity Violation Detection in Concurrent Programs. In: Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2008, 135 – 145.
27. Park S, Lu S, Zhou Y. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems. 2009, 25 – 36.
28. Park S, Vuduc R W, Harrold M J. Falcon: Fault Localization in Concurrent Programs. In: Proceedings of ACM/IEEE International Conference on Software Engineering. 2010, 245 – 254.
29. Biswas S, Huang J, Sengupta A, Bond M D. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014, 28 – 39.
30. Flanagan C, Freund S N. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In: Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2004, 256 – 267.
31. Lu S, Park S, Hu C, Ma X, Jiang W, Li Z, Popa R A, Zhou Y. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In: Proceedings of ACM SIGOPS Symposium on Operating Systems Principles. 2007, 103 – 116.
32. Havelund K. Using Runtime Analysis to Guide Model Checking of Java Programs. In: Proceedings of International SPIN Workshop on Model Checking of Software. 2000, 245 – 264.
33. Cai Y, Wu S, Chan W K. ConLock: A Constraint-based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs. In: Proceedings of International Conference on Software Engineering. 2014, 491 – 502.
34. Eslamimehr M, Palsberg J. Sherlock: Scalable Deadlock Detection for Concurrent Programs. In: Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014, 353 – 365.
35. Sen K. Effective Random Testing of Concurrent Programs. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering. 2007, 323 – 332.
36. Bianchi F A, Margara A, Pezzè M. A Survey of Recent Trends in Testing Concurrent Software Systems. IEEE Transactions on Software Engineering, 2018, 44(8):747 – 783.
37. Thomson P, Donaldson A F, Betts A. Concurrency Testing Using Schedule Bounding: An Empirical Study. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2014, 15 – 28.
38. Desai A, Qadeer S, Seshia S A. Systematic Testing of Asynchronous Reactive Systems. In: Proceedings of Joint Meeting on Foundations of Software Engineering. 2015, 73 – 83.
39. Fu H, Wang Z, Chen X, Fan X. A Systematic Survey on Automated Concurrency Bug Detection, Exposing, Avoidance, and Fixing Tech-

niques. Software Quality Journal, 2018, 26(3):855–889.
40. Arora V, Bhatia R, Singh M. A Systematic Review of Approaches for Testing Concurrent Programs. Concurrency and Computation: Practice and Experience, 2016, 28(5):1572–1611.
41. Souza S R S, Brito M A S, Silva R A, Souza P S L, Zaluska E. Research in Concurrent Software Testing: A Systematic Review. In: Proceedings of Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. 2011, 1–5.
42. Sewell P, Sarkar S, Owens S, Nardelli F Z, Myreen M O. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. Communications of the ACM, 2010, 53(7):89 – 97.
43. Manson J, Pugh W, Adve S V. The Java Memory Model. In: Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2005, 378 – 391.
44. Godefroid P. Model Checking for Programming Languages Using VeriSoft. In: Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1997, 174 – 186.
45. Musuvathi M, Qadeer S. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. ACM SIGPLAN Notices, 2007, 42(6):446 – 455.
46. Emmi M, Qadeer S, Rakamarić Z. Delay-bounded scheduling. ACM SIGPLAN Notices, 2011, 46(1):411 – 422.
47. Burckhardt S, Kothari P, Musuvathi M, Nagarakatte S. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems. 2010, 167 – 178.
48. Godefroid P, Sen K. Combining Model Checking and Testing. In: Handbook of Model Checking. 2018, 613 – 649.
49. Andoni A, Daniliuc D, Khurshid S. Evaluating the "Small Scope Hypothesis. 2003.
50. Lu S, Park S, Seo E, Zhou Y. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems. 2008, 329 – 339.
51. Qadeer S, Rehof J. Context-bounded Model Checking of Concurrent Software. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2005, 93 – 107.
52. Nagarakatte S, Burckhardt S, Martin M M, Musuvathi M. Multicore Acceleration of Priority-based Schedulers for Concurrency Bug Detection. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2012, 543 – 554.
53. Nistor A, Luo Q, Pradel M, Gross T R, Marinov D. BALLERINA: Automatic Generation and Clustering of Efficient Random Unit Tests for Multithreaded Code. In: Proceedings of International Conference on Software Engineering. 2012, 727–737.
54. Li G, Lu S, Musuvathi M, Nath S, Padhye R. Efficient Scalable Thread-safety-violation Detection: Finding Thousands of Concurrency Bugs during Testing. In: Proceedings of ACM Symposium on Operating Systems Principles. 2019, 162–180.
55. Sen K. Race Directed Random Testing of Concurrent Programs. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2008, 11 – 21.
56. Joshi P, Park C-S, Sen K, Naik M. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2009, 110 – 120.
57. Yuan X, Yang J, Gu R. Partial Order Aware Concurrency Sampling. In: Computer Aided Verification. 2018, 317 – 335.
58. Arpaci-Dusseau R H, Arpaci-Dusseau A C. Operating Systems: Three Easy Pieces. 2018.
59. Davis R I, Cucu-Grosjean L, Bertogna M, Burns A. A Review of Priority Assignment in Real-time Systems. Journal of Systems Architecture, 2016, 65(C):64 – 82.
60. Chen D, Jiang Y, Xu C, Ma X, Lu J. Testing Multithreaded Programs via Thread Speed Control. In: Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018, 15 – 25.
61. Flanagan C, Godefroid P. Dynamic Partial-order Reduction for Model Checking Software. In: Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2005, 110 – 121.
62. Yang Y, Chen X, Gopalakrishnan G. Inspect: A Runtime Model Checker for Multithreaded C Programs. 2008.
63. Kokologiannakis M, Raad A, Vafeiadis V. Effective Lock Handling in Stateless Model Checking, 2019:173:1 – 173:26.
64. Bron A, Farchi E, Magid Y, Nir Y, Ur S. Applications of Synchronization Coverage. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2005, 206 – 212.
65. Wang C, Said M, Gupta A. Coverage Guided Systematic Concurrency Testing. In: Proceedings of International Conference on Software Engineering. 2011, 221 – 230.
66. Burnim J, Elmas T, Necula G, Sen K. CONCURRIT: Testing Concurrent Programs with Programmable State-space Exploration. In: Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism. 2012.
67. Hong S, Ahn J, Park S, Kim M, Harrold M J. Testing Concurrent Programs to Achieve High Synchronization Coverage. In: Proceedings of International Symposium on Software Testing and Analysis. 2012, 210 – 220.
68. Joshi P, Naik M, Sen K, Gay D. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In: Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2010, 327 – 336.
69. Sorrentino F, Farzan A, Madhusudan P. PENELOPE: Weaving Threads to Expose Atomicity Violations. In: Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2010, 37 – 46.
70. Eslamimehr M, Palsberg J. Race Directed Scheduling of Concurrent Programs. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2014, 301 – 314.
71. Huang J, Meredith P O, Rosu G. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implemen-

tation. 2014, 337 – 348.
72. Musuvathi M, Qadeer S. Fair Stateless Model Checking. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2008, 362 – 371.
73. Batty M, Owens S, Sarkar S, Sewell P, Weber T. Mathematizing C++ Concurrency. In: Proceedings of Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2011, 55 – 66.
74. Norris B, Demsky B. CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. ACM SIGPLAN Notices, 2013, 48(10):131 – 150.
75. Musuvathi M, Qadeer S, Ball T, Basler G, Nainar P A, Neamtiu I. Finding and Reproducing Heisenbugs in Concurrent Programs. In: Proceedings of USENIX Conference on Operating Systems Design and Implementation. 2008, 267 – 280.
76. Godefroid P. Partial-order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. 1996.
77. Lu S, Jiang W, Zhou Y. A Study of Interleaving Coverage Criteria. In: Proceedings of Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers. 2007, 533 – 536.
78. Yu J, Narayanasamy S. A Case for an Interleaving Constrained Shared-memory Multi-processor. In: Proceedings of Annual International Symposium on Computer Architecture. 2009, 325 – 336.
79. Huang S, Huang J. Maximal Causality Reduction for TSO and PSO. In: Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. 2016, 447 – 461.
80. Huang J, Luo Q, Rosu G. GPredict: Generic Predictive Concurrency Analysis. In: Proceedings of International Conference on Software Engineering. 2015, 847 – 857.
81. Godefroid P, Klarlund N, Sen K. DART: Directed Automated Random Testing. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation. 2005, 213 – 223.
82. Sen K. Concolic Testing. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering. 2007, 571 – 572.
83. Lu G, Xu L, Yang Y, Xu B. Predictive analysis for race detection in Software-Defined Networ. Sciece China. Information Sciences, 2019, 62(62101):1 – 20.
84. De Moura L, Bjørner N. Z3: An Efficient SMT Solver. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2008, 337 – 340.
85. Kasikci B, Cui W, Ge X, Niu B. Lazy Diagnosis of In-production Concurrency Bugs. In: Proceedings of Symposium on Operating Systems Principles. 2017, 582 – 598.
86. Guo S, Kusano M, Wang C, Yang Z, Gupta A. Assertion Guided Symbolic Execution of Multithreaded Programs. In: Proceedings of Joint Meeting on Foundations of Software Engineering. 2015, 854–865.
87. Bergan T, Grossman D, Ceze L. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. In: Proceedings of ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications. 2014, 491–506.
88. Guo S, Kusano M, Wang C. Conc-ISE: Incremental Symbolic Execution of Concurrent Software. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering. 2016, 531–542.
89. Farzan A, Holzer A, Razavi N, Veith H. Con2colic Testing. In: Proceedings of Joint Meeting on Foundations of Software Engineering. 2013, 37–47.

Dongjie Chen is a second year Ph.D student in Department of Computer Science and Technology at Nanjing University. He received his bachelor degree in computer science and technology from Nanjing University. His research is mainly about multi-threaded programs anslysis.

Dr. Yanyan Jiang is an Assistant Researcher of Nanjing University, China. His research interests include software testing, analysis, and synthesis. He was a recipient of ACM SIGSOFT Distinguished Paper Award, CCF Outstanding Doctoral Dissertation Award, and Microsoft Research Asia Fellowship Award. He is also as a Scientific Committee member of provincial Computing Olympiad contests.

Chang Xu received his doctoral degree in computer science and engineering from The Hong Kong University of Science and Technology, Hong Kong, China. He is a full professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University. He participates actively in program and organizing committees of major international software engineering conferences. His research interests include big data software engineering, intelligent software testing and analysis, and adaptive and autonomous software systems. He is a senior member of the CCF and IEEE, and a member of the ACM.

Xiaoxing Ma received his doctoral degree in computer science and technology from Nanjing University, China. He is a full professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nan-

jing University. His research interests include self-adaptive software systems, software architectures, and quality assurance for machine learning models used as software components. He co-authored more than 100 peer-reviewed conference and journal papers and has served as technical program committee members on various international conferences. He is a member of the IEEE and the ACM.