Probabilistic Concurrency Testing for Weak Memory Programs

Mingyu Gao
m.gao-2@student.tudelft.nl
Delft University of Technology
Delft, Netherlands

Soham Chakraborty
s.s.chakraborty@tudelft.nl
Delft University of Technology
Delft, Netherlands

Burcu Kulahcioglu Ozkan
b.ozkan@tudelft.nl
Delft University of Technology
Delft, Netherlands

ABSTRACT

The Probabilistic Concurrency Testing (PCT) algorithm that provides theoretical guarantees on the probability of detecting concurrency bugs does not apply to weak memory programs. The PCT algorithm builds on the interleaving semantics of sequential consistency, which does not hold for weak memory concurrency. It is because weak memory concurrency allows additional behaviors that cannot be produced by any interleaving execution.

In this paper, we generalize PCT to address weak memory concurrency and present Probabilistic Concurrency Testing for Weak Memory (PCTWM). We empirically evaluate PCTWM on a set of well-known weak memory program benchmarks in comparison to the state-of-the-art weak memory testing tool C11Tester. Our results show that PCTWM can detect concurrency bugs more frequently than C11Tester.

CCS CONCEPTS

• **Software and its engineering** → **Concurrent programming structures**; **Software testing and debugging**.

KEYWORDS

Concurrency, Weak memory, Randomized algorithms, Testing

ACM Reference Format:
Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. 2023. Probabilistic Concurrency Testing for Weak Memory Programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3575729

1 INTRODUCTION

In the multicore era, shared memory concurrency plays a key role in improving performance in these architectures. To program these architectures efficiently, the programming languages are introducing first-class concurrency primitives [4, 6, 12, 18, 19, 33] to provide platform-independent abstractions on the hardware and processors. These concurrency primitives empower programmers to achieve greater performance from the architectures. However, programming with these primitives is often error-prone due to their subtle semantics. More specifically, these primitives, as well as the architectures, exhibit additional behaviors that cannot be explained by traditional thread interleaving semantics, aka sequential consistency (SC). These behaviors are known as weak memory behaviors, and these concurrency models are known as weak memory models.

Concurrency poses a significant challenge to testing and verification approaches, considering the number of possible executions even under interleaving semantics. Verification techniques perform sound analyses, but they scale poorly. On the other hand, testing approaches scale better but lacks soundness. Though concurrency testing lacks soundness in general, it is always desirable to achieve some guarantees on the effectiveness of a testing approach.

The Probabilistic Concurrency Testing (PCT) algorithm [8] is a randomized concurrency testing algorithm for SC programs that provides strong theoretical guarantees on the probability of detecting bugs. The probabilistic guarantees of PCT rely on the notion of *bug depth*, i.e., *the minimum number of ordering constraints* between the concurrent events in a program. Given bug depth d as a test parameter, PCT characterizes the set of executions with d ordering constraints and samples a test execution from that set. Focusing on the executions with a certain bug depth significantly reduces the sample set. Unlike naive random testing algorithms that detect a concurrency bug with a probability that is exponentially low in the number of program events n, PCT guarantees a probability that is exponentially low only in d.

In this scenario, a natural question arises: *can we apply PCT for testing weak memory programs?* We investigate this question in this paper and observe that the theoretical guarantee of the PCT algorithm does not apply to testing weak memory programs. It is because weak memory concurrency relaxes the SC requirements and allows a more extensive set of program behaviors, many of which cannot be produced by any interleaving executions in SC. More specifically, the PCT algorithm builds on the notion of bug depth that is designed for the interleaving semantics of sequential consistency, which does not capture weak memory concurrency.

In this paper, we generalize PCT to address weak memory concurrency and present Probabilistic Concurrency Testing for Weak Memory (PCTWM). For this, we revise the definition of concurrency bug depth and generalize it to capture weak memory concurrency. We define bug depth as the *minimum number of communication relations* between the concurrent events in an execution regardless of their scheduling order. We show that the traditional definition of bug depth under SC corresponds to a specific case of our definition, in which the communication relations correspond to the thread interleavings.

Based on our bug depth definition, we devise the PCTWM algorithm that extends the theoretical guarantees of PCT for weak memory concurrency. Similar to PCT, PCTWM provides a theoretical lower bound on the probability of detecting concurrency bugs that is exponential only in the depth bound d. Different from PCT, which samples a test execution with d *ordering requirements*,

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575729

PCTWM samples a test execution with d *communication relations* between the concurrent program events.

We implemented the PCTWM algorithm on top of C11Tester [32], the state-of-the-art testing framework for weak memory programs. We evaluated its performance in detecting weak memory concurrency bugs on a set of well-known weak memory program benchmarks in comparison to the C11Tester concurrency testing algorithm. Our results show that PCTWM can detect concurrency bugs more frequently than C11Tester.

Outline and Contributions. Section 2 provides the required background on weak memory concurrency and PCT. Section 3 presents an overview of our approach. Section 4 discusses the axiomatic model of weak memory concurrency model which focus in this work. Section 5 presents the PCTWM algorithm. Section 6 provides the details on our experimental evaluation and results.

2 BACKGROUND

2.1 Weak Memory Concurrency

In shared memory concurrency, threads communicate through shared memory accesses. The behaviors of these programs are usually explained by thread interleavings, where shared memory accesses in each thread execute in syntactic order, and threads interleave arbitrarily. This is formally known as sequential consistency (SC) [28]. However, concurrent systems usually exhibit additional program behaviors which cannot be explained by interleaving execution and follow a particular *weak memory concurrency* model.

Consider Program SB as an example, where X and Y are shared variables initialized to zero, and the program has two concurrently running threads.

$$\begin{array}{c} X = Y = 0 \\ \begin{array}{r||l} X = 1; & Y = 1; \\ a = Y; & b = X; \end{array} \\ \texttt{assert}(a == 1 \mid\mid b == 1) \end{array} \tag{SB}$$

No interleaving execution violates the assertion in Program SB as at least one of the writes takes place before the reads. However, various weak memory architectures such as x86 [40] or Arm [2] allow the non-SC outcome $a = b = 0$ and violate the assertion. To program these architectures, programming languages like C/C++ [18, 19] provides platform-independent abstractions which also allow this outcome and various other non-SC outcomes in general.

C/C++ Concurrency [18, 19]. C/C++ has different kinds of accesses that affect the behavior of a shared memory concurrent program. To begin with, C11 introduces atomic accesses of four kinds: load, store, atomic update (RMW) such as compare-and-swap and atomic increment, and memory fence. Each atomic access is attached with a memory order from relaxed (RLX), acquire (ACQ), release (REL), acquire-release (ACQ-REL), sequentially-consistent (SC). In addition, C11 provides shared memory load and store accesses that are not atomic, aka non-atomics (NA). Thus, based on the kind of operation and memory order, we categorize the accesses. For instance, an access is *acquire* if its order is one of ACQ, ACQ-REL, SC. Similarly an access is *release* if its order is one of REL, ACQ-REL, or SC. The release and acquire accesses establish synchronization, for instance, when an acquire read reads from a release write. Going forward, in Section 4 we discuss the formal model of C/C++ concurrency in detail.

2.2 PCT vs. Naive Random Testing.

Probabilistic Concurrency Testing (PCT) [8] is a randomized concurrency testing algorithm designed for SC programs, and it provides strong theoretical guarantees on the probability of detecting concurrency bugs. The key to its design is the notion of *concurrency bug depth*, which is defined as the number of ordering constraints between the concurrent events of a program. Given bug depth d, PCT randomly generates a test execution that encodes a particular ordering of events with d scheduling constraints.

Consider Program P1 running two threads, where all the memory accesses are of SC memory order. The program has an assertion violation if the thread on the right reads $X = k$.

$$\begin{array}{r||l} X = 1; & \\ X = 2; & \texttt{assert}(X != k) \\ \ldots; & \\ X = k & \end{array} \tag{P1}$$

A naive random concurrency testing algorithm chooses the next event to schedule from the set of all enabled events at each scheduling choice. Such an algorithm detects the violation in the example with a probability of only $1/2^k$ where k is the number of scheduling choices. To detect it, it must choose the event in the first thread among the 2 enabled events for all k scheduling choices in the execution.

PCT differs from naive random testing by sampling an execution from the set of executions with d ordering constraints. It guarantees a lower bound on the probability of detecting a bug by a test execution with a probability of at least $1/(tk^{d-1})$ where t is the number of threads, k is the number of program events, and d is the bug depth. While naive random testing can detect a bug with an exponentially small probability in the number of program events, PCT detects it with an exponentially small probability only in the bug depth parameter d.

The assertion violation in Program P1 requires a single ordering constraint $d = 1$, i.e., the assertion statement in the second thread must be executed after the $X = k$ statement in the first thread. Given $d = 1$, PCT samples a test execution out of two $d = 1$ executions: It either chooses a schedule that runs all events in the first thread before the second thread or it chooses to schedule the second thread before the first thread. Therefore, it hits the bug with a probability of $1/2$.

3 OVERVIEW

3.1 A Naive Application of PCT to Weak Memory Concurrency

The probabilistic guarantee of the PCT algorithm [8] on the lower bound of the probability of finding bugs does not apply to weak memory programs. We demonstrate this on a naive application of PCT for testing Program P1 where PCT does not detect the violation with a probability of $1/2$.

Consider the $d = 1$ execution of PCT that schedules all events of the first thread before the second thread. Under weak memory concurrency, this execution does not necessarily hit the bug. Because the read event can read from any write event in the first thread.

The example shows that the behavior of the weak memory programs does not depend on the thread interleavings but on the selection of the write events that the read events get the values from. However, the theoretical guarantee of the PCT algorithm relies on the ordering constraints and the interleaving semantics of sequential consistency. More specifically, it relies on the notion of bug depth that is defined as *the minimum number of scheduling constraints that are sufficient to find the bug* [8].

3.2 Revising Concurrency Bug Depth

The existing notion of bug depth does not capture weak memory concurrency bugs. Consider Program SB. The program exhibits a buggy behavior when both variables a and b load the value 0. The bug does not depend on the scheduling order of the events; it does not manifest under any SC executions of the program.

We revise the notion of concurrency bug depth to capture *thread communication* rather than *thread interleavings*. We define the depth of a concurrency bug as the minimum number of *communication relations* between the concurrent events in an execution. A communication relation between two concurrent events communicates the effects of an event (e.g., writing a value) to another event (e.g., reading that value). For example, the depth of the concurrency bug in Program SB is $d = 0$ since it does not require any communication between its thread events. The program events only access the values of the variables that are available in their thread-local *views*.

Notice that the revised definition of the bug depth *extends* the existing notion, which uses thread interleavings. For the specific case of sequential consistency, a thread interleaving induces a communication relation: the effects of all the write events in a thread are communicated to the other threads at the thread interleavings. For example, the depth of the concurrency bug in Program P1 is $d = 1$ under both notions. Under SC, the bug exposes when the execution meets a single ordering constraint, i.e., when the assertion statement is executed after the $X = k$ statement. Under weak memory concurrency, the bug exposes in the presence of a single communication relation between its events, i.e., the communication of the effect of the write $X = k$ to the read event in the assertion in the second thread.

3.3 PCTWM: PCT for Weak Memory

Here, we informally introduce the key ideas in the PCTWM algorithm, which we will elaborate in Section 5.

PCTWM extends PCT to generate an execution with d communication relations instead of d ordering constraints. Bounding the number of communication relations by d restricts the amount of thread interaction in an execution. Without any restrictions, a read operation in a thread can potentially read from a write event in any thread. However, bounding an execution to have only d communication relations allows only d events to read from an external value. The other program events read from their thread-local views, which only keep the updates made available to their threads.

For example, the $d = 0$ execution of Program SB does not allow any load operations to read an external value. Therefore, both load operations read the values available in the local views of their respective threads. Similarly, the $d = 0$ execution of Program P1 restricts the load operation to read the initial value of X. Alternatively, a $d = 1$ execution of the program allows the load operation to read a value written by the remote thread.

Besides the number of communication relations d, PCTWM further parametrizes the execution space using a history depth bounding parameter h. The history depth bound restricts the set of store operations that a load operation can read from based on how *old* a value is. It serves to prioritize the executions that load possibly stale values but not older than h number of store operations. Hence, a load operation that is chosen to form a communication relation can read from only h possible values instead of k values, further reducing the sample set of executions.

For example, a PCTWM execution of Program P1 with $d = 1$ and $h = 2$ detects the concurrency bug with probability 1/2. First, it chooses an event as the *sink* of the $d = 1$ communication relation. This example has only one possible communication sink, i.e., load operation in the assertion statement. The PCTWM algorithm ensures that the selected communication sink event is executed as late as possible, after the execution of other events. In this example, it ensures that the assertion statement is executed after all the events in the first thread, regardless of the initial thread priorities. While the algorithm executes the selected sink event, it chooses a source operation for the communication relation within a history depth $h = 2$. In this example, it can select to read from either $X = (k - 1)$ or $X = k$, each with the probability of 1/2, the latter hitting the bug.

We provide the formal definitions for a communication relation, source and sink events, thread-local view, the complete PCTWM algorithm, its theoretical guarantee, and some example test executions generated by PCTWM in Section 5.

4 WEAK MEMORY CONCURRENCY MODEL

In this section, we discuss the C11 axiomatic model that we will use to formally define the communication relation, which is a core concept of PCTWM.

In C11 axiomatic semantics, a program is represented by a set of executions. An execution consists of a set of events resulting from shared memory accesses or fences and the relations between these events.

Event. An event is represented by a tuple $\langle \mathsf{id}, \mathsf{tid}, \mathsf{lab} \rangle$ where id, tid, and lab denote a unique identifier, thread identifier, and label of the event, respectively. A label $\mathsf{lab} = \langle \mathsf{op}, \mathsf{loc}, \mathsf{rVal}, \mathsf{wVal} \rangle$ is a tuple where op denotes the corresponding memory access or fence operation.

For memory accesses, loc, rVal, and wVal denote the corresponding memory location, the read, and the written value. In case of fences, $\mathsf{loc} = \mathsf{rVal} = \mathsf{wVal} = \perp$. A successful read-modify-write operation results in an RMW event (U) and, on failure, generates a read event (R). The set of read, write, RMW, and fence events are R, W, U, and F, respectively. We write $\mathcal{R} = \mathsf{R} \cup \mathsf{U}$ to denote read or RMW, and $\mathcal{W} = \mathsf{W} \cup \mathsf{U}$ to denote the read or RMW events. The memory locations are initialized at the start of the execution, represented by a set of write events.

Relation. Various binary relations connect the events in an execution. We discuss the notations before explaining them.

Notations. Given a binary relation B, we write $B^?$, B^+, B^*, B^{-1} to denote its reflexive, transitive, reflexive-transitive closures, and inverse relations, respectively. Relation $\mathsf{imm}(B)$ denotes the immediate relation: $\mathsf{imm}(B)(x,y) \triangleq B(x,y) \land \nexists z\, B(x,z) \land B(z,y)$. Given two relations B_1 and B_2, we denote their composition by $B_1; B_2$. $[A]$ denotes the identity relation on a set A, i.e. $[A](x,y) \triangleq x = y \land x \in A$. Given a set S and a relation B, $\mathsf{maximal}(S,B)$ denotes the B-maximal events i.e. $\mathsf{maximal}(S,B) \triangleq \{e \mid e \in S \land S \cap [\{e\}]; B = \emptyset\}$.

An execution has the following relations between events: The relation program-order (po) is a strict partial order that captures the syntactic order between the events. It is a strict total order on same-thread events. Relation reads-from (rf) relates a write event with the same-location read events that read from it. A read event reads from *exactly* one write event. Relation modification-order (mo) is a strict total order over same-location write events. Relation SC is a total order on the SC accesses. From these relations, we derive the following relations.

- From-read ($\mathsf{fr} \triangleq (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus [\mathsf{E}]$) relates a same-location read and write events; if a read r reads-from a write w and write w' is mo-after w, then $\mathsf{fr}(r, w')$ holds.
- We adopt the synchronizes-with (sw) relation from RC20 [34]. Relation happens-before (hb) is the transitive closure of po and sw relations.

$$\mathsf{sw} \triangleq [\mathsf{E}_{\sqsupseteq \mathrm{REL}}]; ([\mathsf{F}]; \mathsf{po})^?; \mathsf{rf}^+; (\mathsf{po}; [\mathsf{F}])^?; [\mathsf{E}_{\sqsupseteq \mathrm{ACQ}}]$$
$$\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{sw})^+$$

Execution. An execution $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{SC}\rangle$ is a tuple where $\mathsf{X.E}$ is the set of events and $\mathsf{X.po}$, $\mathsf{X.rf}$, $\mathsf{X.mo}$, $\mathsf{X.SC}$ are set of po, rf, mo, SC relations between the events in $\mathsf{X.E}$. We represent execution by an *execution graph* where the events are represented by nodes, and the relations are represented by corresponding edges.

Consistency Axioms. C11 defines a set of axioms to check if an execution is consistent.

- (coherence) The events accessing the same memory location are coherent. We categorize them in write and read coherence constraints [26].
  - $\mathsf{mo}; \mathsf{rf}^?; \mathsf{hb}^?$ is irreflexive. (write-coherence)
  - $\mathsf{fr}; \mathsf{rf}^?; \mathsf{hb}$ is irreflexive. (read-coherence)

  These constraints effectively enforce sc-per-location, a total order on same-location memory accesses.
- (Atomicity) The RMW accesses execute atomically. As a result, $(\mathsf{fr}; \mathsf{mo}) = \emptyset$ holds.
- (irrMOSC) The mo and SC orders agree on same-location accesses, that is, $(\mathsf{mo}; \mathsf{SC})$ is irreflexive.
- (SC) The SC accesses are globally ordered. There is a number of SC order definitions [4, 5, 27, 29, 32, 48].
  We follow the one from C11Tester [32], that is, $(\mathsf{hb} \cup \mathsf{rf} \cup \mathsf{SC})$ is acyclic.
  Note that the (SC) axiom enforces that hb is irreflexive (an action cannot *happens-before* itself) [5, 48]. Moreover, as $\mathsf{po} \subseteq \mathsf{hb}$, the (SC) constraint also enforces that $(\mathsf{po} \cup \mathsf{rf})$ is acyclic and forbids out-of-thin-air reads.

5 PCT FOR WEAK MEMORY PROGRAMS

The PCTWM algorithm extends PCT to weak memory programs in a *memory model agnostic* way so that its theoretical guarantee applies to any memory model. The algorithm relies on the two key concepts of (i) *communication relation* between concurrent program events and (ii) *local thread view* that maintains the set of updates made available to a thread. This paper defines and constructs these relations for the C11 memory model.

5.1 Formal Definitions

Definition 1 (View). *A* view *is a map from locations to a set of maximal-*mo *events. Given an execution* $\langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{SC}\rangle$, $\mathsf{view}(x) = \mathsf{maximal}_{\mathsf{mo}}(\mathsf{E}_x)$ *holds where* E_x *are the set of write or RMW events.*

- *Combining views on a location x. We write* $\bigsqcup_{\mathsf{mo}}(\mathsf{view}_1(x), \mathsf{view}_2(x))$ *to compute the maximal view from* $\mathsf{view}_1(x)$ *and* $\mathsf{view}_2(x)$ *for a given location x, i.e.* $\mathsf{maximal}(\mathsf{view}_1(x) \cup \mathsf{view}_2(x), \mathsf{mo})$.
- *Combining views on all memory locations. Similarly, we write* $\bigsqcup_{\mathsf{mo}}(\mathsf{view}_1, \mathsf{view}_2)$ *to compute* $\bigsqcup_{\mathsf{mo}}(\mathsf{view}_1(x), \mathsf{view}_2(x))$ *for all memory locations x.*

Each thread maintains its own view in an execution. We write $t.\mathsf{view}$ *to denote the view of thread t. Essentially, a thread view maintains the latest write or RMW events observed by the thread for each memory location.*

Definition 2 (Communication Relation). *Following the (SC) constraint in C11Tester model (see section 2), we consider inter-thread* rf, hb, SC *as* com *relations, that is,* $\mathsf{com} \triangleq (\mathsf{rf} \cup \mathsf{hb} \cup \mathsf{SC}) \setminus \mathsf{po}$.

Definition 3 (Communication event). *A communication relation is formed between two events: a* source *event and a* sink *event of the communication relation. A* source *event captures the effect, which can potentially be communicated to other threads. So, it is an* SC*, or a write or a fence event. A* sink *event* communicates *the updates of other threads to its local thread. We call the sink events as* communication events. *So, it is an* SC*, or read or acquire event.*

Intuitively, the effect of the events in $\mathsf{dom}(\mathsf{com})$ (e.g., writing a value to a variable, releasing a fence) can potentially be communicated to an event in $\mathsf{codom}(\mathsf{com})$ (e.g., reading the value of a variable, acquiring a fence) running on another thread. We call the events in $\mathsf{dom}(\mathsf{com})$ as communication sources and the events in $\mathsf{codom}(\mathsf{com})$ as communication sinks.

Definition 4 (Bug depth). *The depth of a concurrency bug is the minimum number of* communication relations *between the concurrent events in an execution that is sufficient to produce the bug.*

Definition 5 (History depth). *The history depth h bounds the behavior of a read event in an execution so that it reads from an event that does not have more than h* $\mathsf{imm}(\mathsf{mo})$*-related successors.*

5.2 The PCTWM Algorithm

The PCTWM algorithm randomly generates a test execution with h-bounded d communication relations between the program events. The generated test execution allows d selected events to observe h-bounded updates of external threads and restricts the other events to access only the values in their thread views.

```
Input: $k_{com}$: the number of comm. events in the program
Input: $d$: bug depth
Input: $h$: history depth
Data: threads // the list of threads in ascending order of
      priorities, the first $d$ positions are initially empty
Data: $[d_1, \ldots, d_d]$ // list of $d$ distinct integers, initialized
      randomly between $[1, k_{com}]$
Data: reordered // the set of event ids reordered with a
      thread priority change, initially empty
Data: $i$ // the number of comm. events observed, initially 0
Data: $s$ // the current execution state, initially $s0$
1  Procedure PCTWM($k_{com}, d, h$)
2      while enabled($s$) not empty do
3          for $th \in$ enabled($s$) and $th \notin$ threads do
4              $t \leftarrow$ getHighestPrEnabled(threads)
5              $e \leftarrow$ next($s, t$)
6              if isCommunicationEvent($e$) then
7                  $i \leftarrow i + 1$
8                  if $i \in \{d_1, \ldots, d_d\}$ then
9                      // update the priority of $t$
10                     $k \leftarrow$ indexOf($i, [d_1, \ldots, d_d]$)
11                     threads$[d - k] \leftarrow t$
12                     reordered $\leftarrow$ reordered $\cup \{e\}$
13                     continue
14             executeAndUpdateView($s, e$)
15 Procedure isCommunicationEvent($e$)
16     return $e \in (\mathsf{SC} \cup \mathcal{R} \cup \mathsf{F}_{\sqsupseteq \mathsf{ACQ}})$
```

Algorithm 1: The PCTWM algorithm

```
1  Procedure executeAndUpdateView($t, e$)
2      $b \leftarrow \bot$
3      $x \leftarrow e$.loc
4      if $e \in \mathcal{W}$ then
5          $t$.view($x$) $\leftarrow e$
6      if $e \in \mathsf{SC}$ then
7          $e' \leftarrow$ getSC($t, e$)
8          $t$.view $\leftarrow \bigsqcup_{s.\mathsf{mo}}(t$.view$, e'$.bag$)$
9      if $e \in \mathcal{R}$ then
10         if $e \in$ reordered then
11             // read from any of the store operations
12             $b \leftarrow$ readGlobal($t, h$)
13             if isSync($e, b$) then
14                 $t$.view $\leftarrow \bigsqcup_{s.\mathsf{mo}}(t$.view$, b$.bag$)$
15             else
16                 $t$.view($x$) $\leftarrow \bigsqcup_{s.\mathsf{mo}}(t$.view$(x), b$.bag$(x))$
17         else
18             // read from the local thread view
19             $b \leftarrow$ readLocal($t$)
20     if $e \in \mathsf{F}_{\sqsupseteq ACQ}$ then
21         esw $\leftarrow$ getSWSet($t, e$)
22         for $e' \in$ esw do
23             $t$.view $\leftarrow \bigsqcup_{s.\mathsf{mo}}(t$.view$, e'$.bag$)$
24     if $e \in \mathsf{F}_{REL}$ then
25         // no update to the current thread's view
26     $e$.bag $\leftarrow t$.view
27     $s \leftarrow$ execute($s, t, b$)
```

Algorithm 2: The executeAndUpdateView procedure that executes an event e and updates its thread's view.

Generating a test execution for a weak memory program requires (i) selecting the next event to execute and (ii) selecting the behavior of this event (e.g., selecting which event to read from). The PCTWM algorithm binds these two choices and restricts an execution to switch threads only at d points that correspond to the external reads or synchronization of the inter-thread events.

We present the PCTWM algorithm (see Algorithm 1) following the structure of the C11Tester [32] by (i) incorporating d-bounded test generation in PCT [8], and (ii) maintaining the thread-local views for computing the behavior of communication events.

The PCTWM algorithm takes the bug depth (d), the history bound (h), and the number of communication events in the program (k_{com}) as test parameters. Then, it samples h-bounded d communication relations in the execution.

PCTWM maintains a list of *threads* that keeps the thread ids in the order of their priorities. It chooses the next event to be scheduled using the priority-based approach in PCT. It runs *threads* in the order of their priorities and switches between them at randomly selected d points in the execution. The switching points are specified by the randomly selected tuple of d events, $[d_1, \ldots, d_d]$, randomly initialized between $[1, k_{com}]$. The execution of the selected d events is delayed by updating the thread priorities accordingly, and they are used to form communication relations as they can read from externally written values of the accessed variables. We also keep *reordered*, which keeps the set of events whose execution is delayed. The algorithm variables i and s keep the current number of communication events and the execution state, respectively.

Similar to C11Tester, we use enabled(s) to denote the set of all threads enabled in state s, and next(s, t) to refer to the next enabled event in thread t at state s. We use getHighestPrEnabled(*threads*) to get the thread id with the highest priority among *threads*, and indexOf(i, *list*) to get the index of the element i in *list*.

***Procedure* PCTWM.** The algorithm selects the enabled thread t with the highest thread priority (line 4) and the next enabled event e of t (line 5). If the event is a *communication event*, it is potentially involved in one of the d communication relations. In that case, we increment the number of communication events encountered in the execution (line 7) and check if that event is among the randomly selected d events (line 8). If this is the case, we delay the execution of its thread by updating its priority (line 11) and adding the event to the set *reordered* (line 12). On lines 10-11, we update the priority of the current thread based on event e's index in the tuple $[d_1, \ldots, d_d]$. Suppose e is identified by $i = d_e$, in $[d_1, \ldots, d_d]$. We delay the execution of e so that it executes after all program events except for the events in $[d_{e+1}, \ldots, d_d]$. Hence, the algorithm runs the communication events identified by $[d_1, \ldots, d_d]$ as late as possible, preserving their relative order in the tuple. Enforcing a particular order between these d communication events provides the visibility of a communication source (dom(com)) event to its sink (codom(com)) event.

The PCTWM algorithm is agnostic to any memory model by using the two procedures:

- isCommunicationEvent, which is used to check if an event is a *communication event* which is potentially delayed to form a communication relation (line 6), and
- executeAndUpdateView, which updates the local views of the thread based on the executed event e (line 14).

In this work, we define these procedures based on the C11 memory model (described in Section 2), which is also the considered model by C11Tester.

Procedure isCommunicationEvent. Given an event e, the procedure checks if it is a communication event. Following the definition of communication events in Section 5.1, a communication event is: (1) an SC event or (2) a read event, which may read from other threads, or (3) a synchronization event, which can be a sink of an inter-thread synchronization (sw) relation.

Procedure executeAndUpdateView. Given the scheduled event e and its thread t, this procedure executes e and updates the thread-local view of t accordingly. For each event, we maintain a bag that captures the thread-local view at the point of its execution. Whenever an event forms a communication relation where it is the source event, we communicate its bag to the sink event of the communication relation. The sink event uses the bag to update its own thread-local view.

The update depends on the communication relation formed between the events. On line 2 in Algorithm 2, we keep a reference b for a read or RMW event e to store the behavior of the write event e reads-from. We update the view of thread t and the bag of e based on the type of e.

On lines 4-5, if e is a write or RMW event, then the view of t is updated only with event e at the location of e i.e. x. On lines 6-8, if e is an SC event, then the algorithm updates the view of thread t with the views of that event's SC-predecessors (returned by getSC). Lines 9-19 handle the read events. If e in the *reordered* set, i.e., it is selected as one of the communication sinks, then e reads from a visible write or RMW event b within history depth h using readGlobal (line 12). Otherwise, it reads from the value from its thread's local view using readLocal (line 19). The readGlobal procedure forms a communication relation between e and the operation it reads from i.e., b. Depending on the communication relation e forms, it can either form a sw relation (checked by isSync) and, therefore, updates its thread view by all the variables of the bag it receives from the communication source (line 14), or it only updates its thread view by only the value of e.loc it reads (line 16). Lines 20-23 handle the synchronization formed if e is an $\mathsf{F}_{\sqsupseteq \text{ACQ}}$ event. It updates the thread's view on all locations with the bags of all events with which it synchronizes-with. On line 25, if the operation is an F_{REL}, then nothing is communicated to the current thread and the thread view is not updated. Finally, on line 26, The algorithm assigns the current thread's view to the bag of the currently executing event e and executes the event.

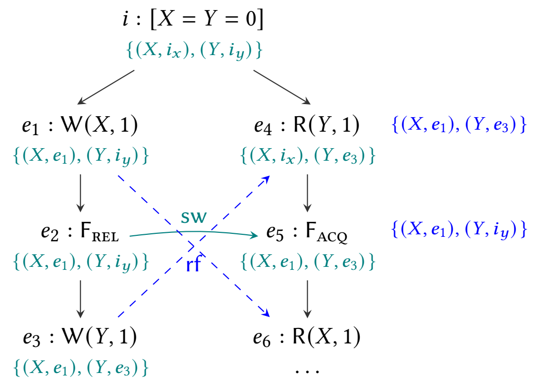

Figure 1: MP1 execution $a = 1, b = 1$ with views and bags.

Example. Consider the Program MP1 running two threads $T1$ and $T2$. In this program, $a = 1, b = 0$ results in a bug.

$$\begin{array}{r||l} \multicolumn{2}{c}{X = Y = 0} \\ X_{\text{RLX}} = 1; & a = Y_{\text{RLX}}; \\ \mathsf{F}_{\text{REL}}; & \mathsf{F}_{\text{ACQ}}; \\ Y_{\text{RLX}} = 1; & b = X_{\text{RLX}}; \end{array} \tag{MP1}$$

The execution in Figure 1 demonstrates how the algorithm ensures that if $a = 1$, then also $b = 1$. At the beginning of the execution, the initial views of the threads $T1, T2$ are $\{(X, i_x), (Y, i_y)\}$, where i_x and i_y are initialization writes of X and Y respectively. The execution of e_1 updates the thread view to $\{(X, e_1), (Y, i_y)\}$, which remains the same after e_2 following lines 4-5 and 25 in Algorithm 2, respectively. Execution of e_3 updates the thread view on Y (lines 4-5). The read event e_4 reads from e_3 (following line 12) and obtains $T1$'s view in its bag (we illustrate the communicated bags using blue-colored views). The relaxed read operation updates the thread view only for Y, resulting in $\{(X, i_x), (Y, e_3)\}$ (line 16). Fence event e_5 synchronizes with e_2 and obtain $\{(X, e_1), (Y, i_y)\}$ in its bag to update $T2$'s view to $\{(X, e_1), (Y, e_3)\}$ following lines 20-23, which overwrites the initialization on X. The next event e_6 reads the value 1, regardless of whether it reads using readGlobal or readLocal. Because its current thread view keeps e_1 for the variable X. In that example, the outcome $a = 1, b = 0$ triggers a bug.

5.3 Example Executions Generated by PCTWM

We now discuss some example executions generated by PCTWM for testing Program MP2, which is a message-passing program in which all the shared memory accesses are relaxed accesses. The program consists of the parallel execution of three threads, which we refer to as $T1$, $T2$, and $T3$, from left to right. The execution of a program that reads $Y == 1$ and $X == 0$ in $T3$ hits an assertion violation. While proper synchronization of the operations could prevent the assertion violation, we consider this buggy version of the program with all relaxed accesses to illustrate the test case generation of PCTWM and how it detects the bug. The bug has concurrency bug depth $d = 2$ since it exposes in an execution with two communication relations between its threads.

We present three test executions generated by PCTWM with $d = 0$, $d = 1$, and $d = 2$ communication relations, respectively.

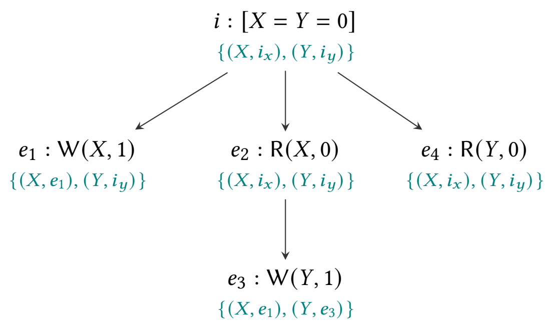

Figure 2: The $d = 0$ execution of Program MP2. There is no communication between the threads.

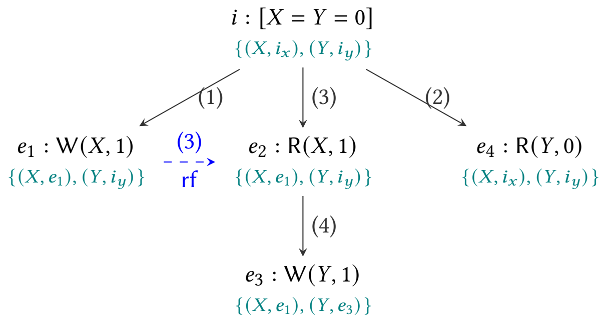

Figure 3: A $d = 1$, $h = 1$ execution of MP2. The illustrated execution selects $[e_2]$ as the sink of the communication relation and assigns initial priorities as $[T1, T2, T3]$.

$$
X = Y = 0
$$
$$
X = 1; \;\left\|\; \begin{array}{l} \textbf{if}(X == 1) \\ \quad Y = 1; \end{array} \;\right\|\; \begin{array}{l} \textbf{if}(Y == 1) \\ \quad \textbf{if}(X == 0) \\ \qquad \textbf{assert}(\text{false}); \end{array} \qquad \text{(MP2)}
$$

Generating the execution with $d = 0$. The $d = 0$ execution of Program MP2 (see Figure 2) does not have any communication relations between the threads. Therefore, all the events access the values in their thread local views. In the figure, we provide the thread views (below the events) obtained after executing an event.

Following Algorithm 1, PCTWM generates this execution by assigning random priorities to the threads and running them serially in the order of their priorities. Given $d = 0$, it does not update priorities at any point in execution and does not introduce any communication relations into the execution.

Generating an execution with $d = 1$. The PCTWM algorithm generates a $d = 1$ execution of the program by randomly sampling a communication relation in the execution. Figure 3 provides a $d = 1$ and $h = 1$ execution of the program with randomly assigned initial thread priorities $[T1, T2, T3]$, respectively, in the decreasing order.

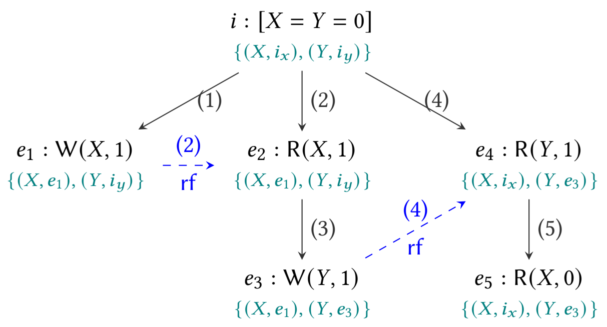

Figure 4: A $d = 2$, $h = 1$ execution of MP2. The illustrated execution selects $[e_2, e_4]$ as the sinks of the two communication relations and assigns initial priorities as $[T1, T2, T3]$.

We mark the execution order of the events with the numbers on the arrows.

Given $d = 1$, the algorithm switches the execution of threads at a randomly selected communication event, allowing that event to read from a value written in another thread (e.g., a read event can read from an external write event) or synchronize with an external event (e.g., a read-acquire event can synchronize with a write-release event, or an SC event can synchronize with another SC event). In the example execution, the algorithm selects $[e_2]$ as the sink of the communication relation.

The execution starts with running the highest priority thread, $T1$. It runs e_1 and moves to the next thread, $T2$. Since the next event, e_2, is selected as the sink event of the communication relation, PCTWM does not immediately run this event. It reduces the priority of $T2$ to a value smaller than the initial thread priorities. This causes the communication sink event e_2 to run after all other events, e.g., possibly write events it can externally read from. The algorithm continues with the currently highest priority thread, $T3$. The event e_4 reads $Y = 0$, so the execution does not go into the if branch in the program. After the completion of $T3$, the algorithm resumes $T2$, running the event e_2. Given $h = 1$, e_2 reads from the last written value of X from e_1, forming a communication relation from e_1 to e_2 between the threads $T1$ and $T2$. The execution continues with running e_3 and completes.

In an alternative execution with $h = 2$, PCTWM would select one of the writes $X = 0$ or $X = 1$ uniformly at random for e_2 to read from, reading from either the initial value or e_1, respectively.

Generating an execution with $d = 2$. PCTWM generates a $d = 2$ execution of the program by randomly sampling two communication relations in the execution. Figure 4 provides a $d = 2$ and $h = 1$ execution of the program with randomly assigned initial thread priorities $[T1, T2, T3]$, respectively, in the decreasing order. Different from the previous example, a $d = 2$ execution switches threads at two randomly selected events, allowing these two events to read from or synchronize with an external event. In this example, PCTWM selects the tuple $[e_2, e_4]$ to execute in this order after the execution of all other events and form communication relations accessing thread external writes.

The execution starts similarly to the previous example. PCTWM runs the highest priority thread $T1$ and continues with $T2$. Since the next event, e_2, is selected as the sink event of the communication relation, PCTWM does not immediately run this event. It reduces the priority of $T2$ and moves to $T3$. Since the next event, e_4, is also selected as the sink of a communication relation, PCTWM does not immediately execute e_4, but it reduces the priority of $T3$. Note that PCTWM updates the priorities of $T2$ and $T3$ so that e_2 and e_4 run in the order they appear in $[e_2, e_4]$. Therefore, it runs the selected events $[e_2, e_4]$ in that order regardless of their random initial thread priorities. The execution continues with the current highest priority thread, $T2$, allowing e_2 to read from an external write. Given $h = 1$, it reads from e_1, forming a communication relation. After the completion of $T2$, the algorithm resumes $T3$. The event e_4 forms the second communication relation by reading from e_3. Since the relaxed read operation updates only the thread-local view of the thread for Y but not for X, e_5 reads $X = 0$. This execution with $d = 2$ communication relations produces a buggy execution where $T3$ reads $Y = 1$ and $X = 0$.

The example test executions of Program MP2 highlight several insights about the PCTWM algorithm. First, more complex executions with deeper concurrency bugs manifest in the existence of a higher number of communication relations between concurrent events. Second, the execution order of the selected d events affects the set of visible values to a read event to read from. For example, if the algorithm generates a $d = 2$ execution by selecting $[e_4, e_2]$ instead of $[e_2, e_4]$, then e_4 reads $Y = 0$, resulting in an execution that does not produce the bug. Finally, communication relations update thread local views based on the semantics of the events in the relation. For example, the communication relation (e_3, e_4) in Figure 4 updates only the variable Y in the thread local view of $T3$. However, if the communication relation (e_3, e_4) formed a synchronization (e.g., e_3 was a release-write and e_4 was an acquire-read), the updates on both variables X and Y would be propagated to the thread local view of $T3$.

5.4 The Probability of Detecting Bugs

Given a program with k_{com} communication events, PCTWM samples an execution with d communication relations with a history depth of h with the probability of at least $1/O((hk_{com})^d)$. The algorithm chooses d events out of k_{com} events as the sinks of d communication relations from $\binom{k_{com}}{d}$ possible ways. It sorts these d events in a particular order yielding $\binom{k_{com}}{d}d! \leq {k_{com}}^d$ many ways. For each of the d communication sinks, the algorithm selects a source event out of h possible events in $O(h^d)$ possible ways. Therefore, the size of the set of executions sampled by the PCTWM algorithm is bounded by $O((hk_{com})^d)$. Trivially, the probability of choosing an execution out of this set is at least $1/O((hk_{com})^d)$, which is exponentially low only in the bug depth parameter d.

6 EXPERIMENTAL EVALUATION

In this section, we discuss our evaluation of PCTWM on some well-known data structures and real-world application benchmarks and compare the results with the state-of-the-art weak memory testing tool, C11Tester.

Table 1: Data structure benchmarks.

Benchmark	LOC	k	k_{com}	d
dekker	50	20	14	0
msqueue	232	49	31	0
barrier	38	15	10	1
cldeque	122	86	56	1
mcslock	75	26	16	1
mpmcqueue	108	19	17	2
linuxrwlocks	90	20	19	2
rwlock	98	84	74	2
seqlock	50	20	18	3

Random Testing in C11tester [32]. C11Tester randomly explores the set of program behaviors by generating test executions in two steps: (1) randomly selecting the next thread to execute among the set of all enabled events and (2) randomly selecting a write from a set of visible writes for a read or update to read-from.

Implementation. We developed both PCT and PCTWM algorithms in the C11Tester framework, which provides interfaces for implementing the selection of next event to execute, and the selection of program behavior for a read event to read from. Our implementation of PCT differs from the original algorithm [8], as our implementation does not produce only sequentially consistent executions, but it allows for some weak memory behavior. More specifically, we implement a variant of PCT where the read operations do not necessarily read the last written value on a variable, but they read any of the observable values under the given memory model, selected uniformly at random. Therefore, our implementation of PCT can trigger the concurrency bugs that occur only under weak memory behavior.

Benchmarks. We use nine data structure benchmarks (with some seeded weak memory concurrency bugs by C11Tester), and three real-world applications Iris [50], a low-latency C++ logging library; Mabain [13], a key-value store library; Silo [46, 47], a multi-core in-memory storage engine used in the evaluation of C11Tester.

Table 1 lists the benchmarks together with their lines of code (LOC), the estimated number of program events (k), the estimated number of communication events (k_{com}), and the depth of their concurrency bugs (d). Similar to PCT, which takes an estimated number of program events (k) and bug depth d as test parameters, PCTWM takes an estimated number of communication events (k_{com}) together with the bug depth d and history depth h as test parameters.

Research Questions. We evaluate the effectiveness of PCTWM by addressing the following research questions:

RQ1. Can the PCTWM algorithm find concurrency bugs and how do the bounding parameters bug depth (d) and history depth (h) affect the bug detection rate?

RQ2. How does the bug detection rate of the PCTWM algorithm compare to the bug detection rate of C11Tester?

RQ3. How does the effectiveness of PCTWM compare to PCT for testing programs having more weak memory accesses?

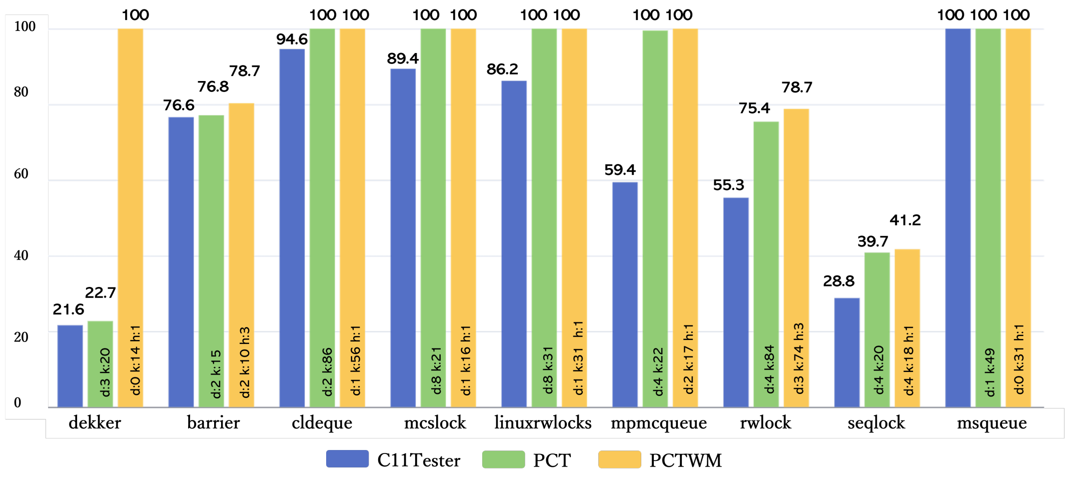

Figure 5: Highest bug hitting rates observed for all nine benchmarks

RQ4. What is the performance overhead of PCTWM in terms of execution time in comparison to C11Tester?

6.1 The Effectiveness of PCTWM

Varying the bug depth bound. To answer RQ1, we first test each benchmark with a d value as an algorithm input parameter that corresponds to the depth of their concurrency bugs. Next, we vary the bounding test parameters d and h to observe their effect on the effectiveness of hitting the bugs.

Table 2 lists the percentages of the test executions that detect the bug out of 1000 test runs. PCTWM successfully detects the bugs with high probabilities by bounding the sample set of executions using varying values $[d, d+2]$ for the bug depth parameter.

For the benchmarks having a concurrency bug of depth $d = 0$, PCTWM generates a single execution that does not introduce any communication relations and detects the bug in all tests. We also see that increasing values for the bug depth parameter d decreases the probability of hitting the bug for these benchmarks. For the other benchmarks, PCTWM detects the bugs with comparable rates for the depth parameter values $[d, d+2]$. We observe that the rate of detecting bugs decrease for larger values of d [16].

Varying the history depth bound. Table 3 lists the percentages of the test executions that detect the bugs for varying values of the history bound $h = [1, 4]$. We observe only small changes in the bug detection rates for increasing h. This can be because there are not many visible write events within h bound for a read event to read from in the benchmark programs. The history bounding parameter is more useful for programs with a high number of write events whose values are visible to read events.

Table 2: Bug hitting rates using PCTWM for the data structure benchmarks for varying values of bug depth d.

Benchmark	d	Rate(d)	Rate(d+1)	Rate(d+2)
dekker	0	100 (h:1)	77.1 (h:1)	75.7 (h:1)
msqueue	0	100 (h:1)	100 (h:1)	100 (h:1)
barrier	1	77.8 (h:2)	78.7 (h:3)	75.9 (h:2)
cldeque	1	55.7 (h:3)	100 (h:1)	100 (h:1)
mcslock	1	100 (h:1)	100 (h:1)	100 (h:1)
mpmcqueue	2	100 (h:1)	100 (h:1)	100 (h:1)
linuxrwlocks	2	100 (h:1)	100 (h:1)	100 (h:1)
rwlock	2	76.9 (h:4)	78.8 (h:3)	77 (h:3)
seqlock	3	33.3 (h:3)	41.2 (h:1)	39.5 (h:2)

Table 3: Bug hitting rates using PCTWM for the data structure benchmarks for varying values of history depth h.

Benchmark	k_{com}	d	Bug Hitting Rate(%)			
			h:1	h:2	h:3	h:4
dekker	14	1	77.1	69.7	67.4	65.3
msqueue	31	0	100	100	100	100
barrier	10	2	74.8	75.1	76.7	78.7
cldeque	56	1	100	100	100	100
mcslock	16	1	100	100	100	100
mpmcqueue	17	2	100	100	100	100
linuxrwlocks	19	2	100	100	100	100
rwlock	74	3	74.2	76	78.8	73.5
seqlock	18	4	41.2	39.8	37.1	36.7

6.2 PCTWM vs C11Tester

We compare the performance of random testing using PCTWM, our implementation of PCT, and C11Tester in Figure 5. Note that our implementation of PCT does not restrict the test executions to be sequentially consistent but allows them to exhibit some weak memory behavior. Therefore, it can detect weak memory concurrency bugs in the benchmark programs.

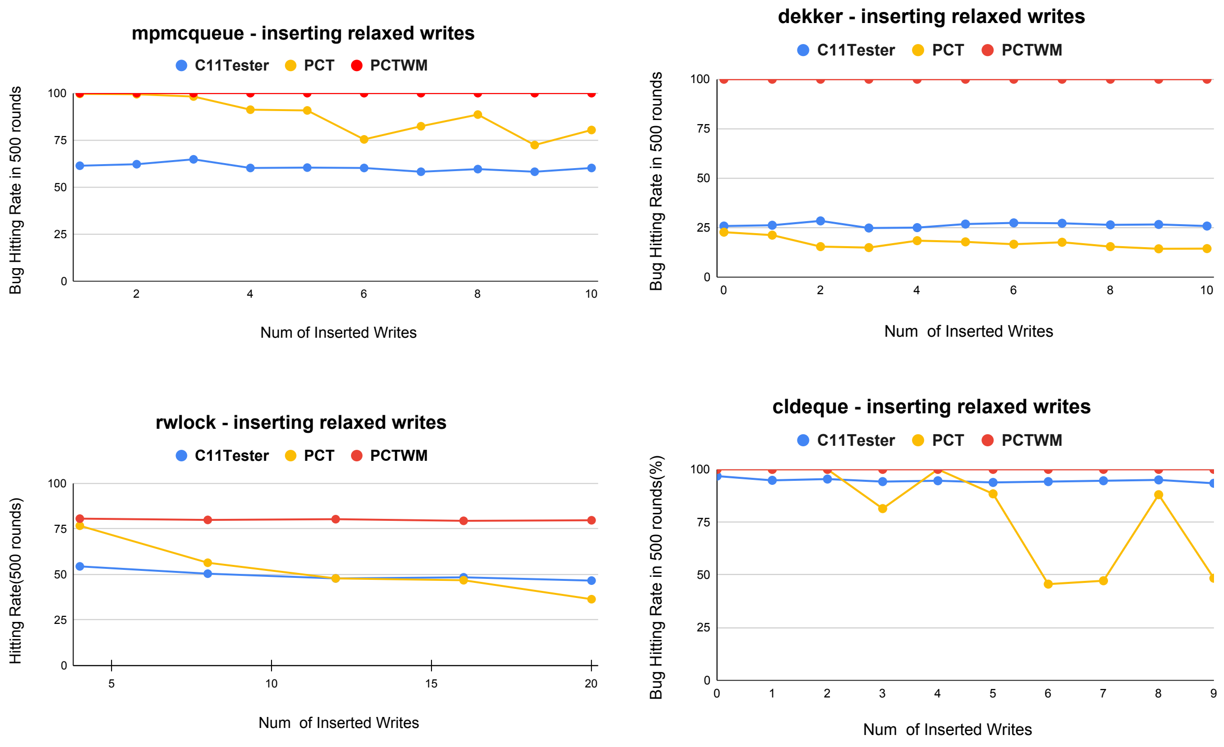

Figure 6: Change in the bug hitting rates with an increasing number of relaxed write operations in the benchmarks.

PCT vs. C11Tester. PCT achieves higher bug-hitting rates than C11Tester in general and performs significantly better in five of the benchmarks. This can be explained by the fact that the PCT algorithm samples a test execution from a d-bounded set of test executions while C11Tester samples from the set of all possible program executions.

PCTWM vs. PCT. The PCTWM algorithm performs comparably or better than PCT in most of the benchmarks. For the benchmarks with a concurrency bug of depth $d = 0$, PCTWM is observably better than PCT. Because the bug in these benchmarks expose when there is no communication between its threads and the PCTWM executions with $d = 0$ always hit them. The PCT and PCTWM algorithms improve the average bug-hitting rate in the nine benchmarks by 16% and 29%, respectively.

In general, the bounded testing algorithms PCT and PCTWM outperform C11Tester except for the seqlock benchmark where they hit the bug with a slightly lower rate. It is because this benchmark implements *wait loops* in which a thread waits for a value written by another thread. PCT and PCTWM restrict the thread interleavings and communication, respectively, preventing the thread from going out of the wait loop. Similar to PCT, which uses some heuristics to avoid such starvation issues [8], PCTWM applies a heuristic to switch to a random thread when it observes a livelock. The more thread switches and external reads-from PCTWM employs to avoid a livelock, the more it approaches to naive random testing.

While the performances of PCT and PCTWM are comparable, PCTWM performs slightly better than PCT. Theoretically, PCTWM improves over PCT for the programs having weaker memory behaviors than SC. Therefore, the observable performance improvement of PCTWM over PCT depends on the amount of weak memory behavior in the benchmark program under test.

6.3 PCTWM vs PCT

The performance improvement of PCTWM over PCT is more observable with increasing relaxed memory operations in the programs. In RQ3, we aim to address how the performance of PCTWM improves over PCT for the programs with a higher amount of weak memory behaviors. To do so, we insert relaxed write accesses in the benchmark programs which do not affect the program behavior or the depth of the concurrency bug. Essentially, we increase the number of program events and the number of visible writes to read-from for the read or RMW accesses.

In Figure 6, we observe significant differences in the bug detection rates of PCT and PCTWM. The bug detection ability of the PCTWM stays stable while that of the PCT fluctuates. This empirical observation aligns with the probabilistic guarantees of PCT and PCTWM. The increased number of program events in the modified benchmarks decreases the probability of detecting bugs with PCT, which selects d events to reorder out of all program events. In contrast, the increased number of relaxed write operations in a program does not affect the performance of PCTWM.

6.4 PCTWM vs C11Tester: Performance Overhead

To answer RQ4, we evaluate the performance of C11Tester and PCTWM on some real-world applications. We tested the applications using both single and multiple CPU cores. Table 4 lists the performance assessment results averaged over 10 runs. We compare the performance results of PCTWM to that of C11Tester using the same measurements used earlier for the evaluation of C11Tester [32]. Accordingly, we report the test throughput (in terms of op/sec) for the Silo benchmark and the elapsed time (in seconds) for the Mabain and Iris benchmarks.

In our experiments, both C11Tester and PCTWM detect data races in all of these applications in single as well as multiple core configurations. Considering individual applications, we do not observe a significant difference in the throughput result in Silo. In Mabain and Iris, the execution time in PCTWM is higher than C11Tester, around 10% and 16%, respectively. This can be explained by the computation overhead in the PCTWM algorithm. PCTWM computes the thread-local views and selects the values to read from based on the local or global writes, whereas C11Tester does not maintain this information and randomly selects a write operation from the set of visible writes. Finally, we observe that the configurations do not affect the performance as the C11Tester framework runs one thread at a time.

Table 4: Performance on testing real-world applications. In parentheses, we include the relative standard deviation.

	core	C11tester	PCTWM
Silo	single	12428 (0.58%)	11039 (7.38%)
(ops/sec)	multiple	11987(0.61%)	11387 (6.92%)
Mabain	single	7.73 (1.56%)	8.43 (4.11%)
(time/s)	multiple	7.65 (2.48%)	8.40 (3.62%)
Iris	single	10.98 (2.02%)	12.79 (4.78%)
(time/s)	multiple	10.83 (1.88%)	12.43 (6.59%)

7 RELATED WORK

Concurrency and consistency. Memory consistency models play a crucial role in concurrent systems. Architectures [2, 3, 39, 42] exhibit weak memory concurrency behaviors due to various architectural features such as memory hierarchy, interconnect and so on for performance reason. To gain performance from these architectures, the high level programming languages also introduce primitives and a number of programming models for weak memory concurrency are defined [5, 6, 11, 20, 21, 25, 27, 33, 34, 41, 48]. In this paper we follow the C/C++ concurrency model [5, 31, 34]. However, due to the subtle semantics of these primitives, writing weak memory concurrent programs are often difficult and error prone. Therefore weak memory concurrency pose a significant challenge to testing and verification.

Testing and verification of weak memory concurrency. In recent years a number of approaches are developed for weak memory verification [1, 3, 22, 38]. Verifying weak memory program is even more challenging as it may require to explore larger set of executions than SC. In this scenario testing [29–31] and dynamic analysis [9, 10, 15] approaches for weak memory concurrency have been effective in handling larger programs while sacrificing soundness.

Concurrency testing. Many algorithms and tools have been proposed for testing the concurrency behavior of programs running under SC.

Systematic testing relies on a controlled scheduler that can enforce a particular ordering of thread events in execution and enumerates test executions for the scheduler. Due to the explosion in the number of possible executions of a concurrent program, testing algorithms focused on exercising a bounded set of program behaviors. These include generating test executions with a bounded number of context switches [43], nonpreemptive contexts [35], scheduler delays [14], and phases [7].

Randomized testing aims at detecting bugs by randomly generated test executions, and they are shown [45] to be effective in practice. The randomized partial order sampling algorithm [44] is designed to cover execution traces more uniformly than pure random walk. The probabilistic concurrency testing (PCT) algorithm [8] improved state-of-the-art by providing a theoretical guarantee on the random testing. The parallel PCT algorithm (PPCT) [36] allows the parallel execution of many threads instead of serializing them. The PCT algorithm for multithreaded programs with a set of totally ordered events is extended to distributed systems [23, 24, 37], to capture a partially ordered set of events. The partial order sampling (POS) algorithm [49] also provides theoretical probability bounds on the generated tests. PCT differs from the other randomized algorithms as it guarantees a probability of detecting bugs that is exponentially low only in the bug depth, d.

8 CONCLUSION

We presented the Probabilistic Concurrency Testing for Weak Memory (PCTWM) algorithm for testing weak memory concurrency programs and provides theoretical guarantees on the probability of detecting bugs. PCTWM extends the Probabilistic Concurrency Testing (PCT) algorithm that is designed for SC programs to capture weak memory concurrency. PCTWM achieves this by (i) revising the existing notion of concurrency bug depth that is defined based on *thread interleavings* to capture *thread communications*, and (ii) devising an algorithm to sample a test execution from the set of program behaviors with a bounded number of thread communication relations.

We implemented PCTWM and evaluated its performance in comparison to the state-of-the-art weak memory program testing tool C11Tester. Our evaluation demonstrates that PCT and PCTWM improve the C11Tester's bug detection ability as they enhance the hitting rate in most of these benchmarks. Moreover, PCTWM outperforms PCT for testing the weak memory programs with more relaxed write operations.

DATA-AVAILABILITY STATEMENT

The artifact is available on Zenodo [17]. Appendix A provides the details on how to use the artifact and how to reproduce the results.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact is in https://doi.org/10.5281/zenodo.7225459. This is a VagrantBox package (~6 GB) containing the artifact for the paper 'Probabilistic Concurrency Testing for Weak Memory Programs'. This vagrant package offers the experimental environment, which contains all code, benchmarks, and scripts to reproduce the experimental results in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** The scripts are offered to reproduce the results of PCT and PCTWM algorithms, implemented on C11Tester. The results of the original C11Tester experiments are listed in the C11Tester paper, and we quote them in our paper.
- **Metrics:** We follow similar evaluation metrics of original C11Tester. The metric for evaluating the bug detection ability of each algorithm is Bug Hitting Rate(%). It refers to the number of hitting the bug in the benchmarks over 1000 rounds or 500 rounds. Average Running time(ms) is evaluated for each data structure benchmark and real applications to show the speed of detecting the bug. Throughout(ops/sec) is another metric when evaluating the real-world applications.

A.3 Requirements

Hardware. The PC or computer should have memory larger than 64 GB and RAM larger than 16 GB.

Software.

- Install VirtualBox 6.1.26 (https://www.virtualbox.org/wiki/Changelog-6.1#v26).
- Install VagrantBox 2.2.18 (https://www.vagrantup.com/intro/v2.2.18).

Running Vagrantbox. To run the artifact's vagrantbox, please execute the commands below:

- It may require to run 'vagrant init package.box' in the artifact root directory first.
- It may require to the add the following command to the Vagrantfile, right before the final 'end'.

```
config.vm.provider "virtualbox" do |v|
v.customize ["modifyvm", :id, "–uartmode1", "disconnected"]
end
```

- 'vagrant up'
- It may require: 'vagrant provision'
- vagrant ssh

A.4 Experimental Workflow

We evaluate the research questions (RQ) as follows:

- (RQ1, RQ2) We compute the bug hitting rate in 1000 runs on nine benchmarks varying the parameters in Appendix A.5.
- (RQ3) We change the nine benchmarks by inserting more 'relaxed write' accesses and compute the bug hitting rate in 1000 runs.
- (RQ4) We compute the throughput and average running time of three real applications, and for nine benchmarks, we compute the average running time.

The experiments can be run by the following python scripts.

- 'result_pctwm.sh' runs the PCTWM experiments,
- 'result_pct.sh' runs the PCTWM experiments, and
- 'run_all.sh' runs both PCT and PCTWM experiments.

The results can be observed in the console with the tips - 'results for questions 1/2/3/4'. We manually plot the results in Figures 5 and 6, and Tables 1 to 4.

A.5 Experiment Customization

The parameters in our algorithms are as follows.

For the PCT algorithm. :

- -b: Bug depth
- -l: Number of shared access events
- -s: Seed number

For the PCTWM algorithm. :

- -d: Bug depth
- -k: Number of communication events
- -y: History depth
- -s: Seed number

REFERENCES

[1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9035)*, Christel Baier and Cesare Tinelli (Eds.). Springer, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28

[2] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 8:1–8:54. https://doi.org/10.1145/3458926

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

[4] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 634–648. https://doi.org/10.1145/2837614.2837637

[5] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66. https://doi.org/10.1145/1926385.1926394

[6] John Bender and Jens Palsberg. 2019. A formalization of Java's concurrent access modes. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 142:1–142:28. https://doi.org/10.1145/3360568

[7] Ahmed Bouajjani and Michael Emmi. 2012. Bounded Phase Analysis of Message-Passing Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7214)*, Cormac Flanagan and Barbara König (Eds.). Springer, 451–465. https://doi.org/10.1007/978-3-642-28756-5_31

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

[9] Jacob Burnim, Koushik Sen, and Christos Stergiou. 2011. Testing concurrent programs on relaxed memory models. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 122–132. https://doi.org/10.1145/2001420.2001436

[10] Man Cao, Jake Roemer, Aritra Sengupta, and Michael D. Bond. 2016. Prescient memory: exposing weak memory model behavior by looking into the future. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management, Santa Barbara, CA, USA, June 14 - 14, 2016*, Christine H. Flood and Eddy Zheng Zhang (Eds.). ACM, 99–110. https://doi.org/10.1145/2926697.2926700
[11] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 70:1–70:28. https://doi.org/10.1145/3290383
[12] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102
[13] Changxue Deng. 2018. Mabain: A fast and light-weighted key-value store library. https://github.com/chxdeng/mabain.
[14] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 411–422. https://doi.org/10.1145/1926385.1926432
[15] Cormac Flanagan and Stephen N. Freund. 2010. Adversarial memory for detecting destructive races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 244–254. https://doi.org/10.1145/1806596.1806625
[16] Mingyu Gao. 2022. *Probabilistic Testing for Weak Memory Concurrency.* Master's thesis. Delft University of Technnology.
[17] Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. 2022. Probabilistic Concurrency Testing for Weak Memory Programs — Artifact. Available at https://doi.org/10.5281/zenodo.7225459.
[18] ISO/IEC 14882. 2011. Programming Language C++.
[19] ISO/IEC 9899. 2011. Programming Language C.
[20] Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM, 759–767. https://doi.org/10.1145/2933575.2934536
[21] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. https://doi.org/10.1145/3009837.3009850
[22] Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 427–440. https://doi.org/10.1007/978-3-030-81685-8_20
[23] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 160:1–160:28. https://doi.org/10.1145/3276530
[24] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. 2019. Trace aware random testing for distributed systems. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 180:1–180:29. https://doi.org/10.1145/3360606
[25] Ori Lahav and Udi Boker. 2022. What's Decidable About Causally Consistent Shared Memory? *ACM Trans. Program. Lang. Syst.* 44, 2 (2022), 8:1–8:55. https://doi.org/10.1145/3505273
[26] Ori Lahav and Roy Margalit. 2019. Robustness against release/acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 126–141. https://doi.org/10.1145/3314221.3314604
[27] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632. https://doi.org/10.1145/3062341.3062352 Technical Appendix Available at https://plv.mpi-sws.org/scfix/full.pdf.
[28] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439
[29] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic race detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 443–457. https://doi.org/10.1145/3009837.3009857
[30] Christopher Lidbury and Alastair F. Donaldson. 2019. Sparse record and replay with controlled scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 576–593. https://doi.org/10.1145/3314221.3314635
[31] Nian Liu, Binyu Zang, and Haibo Chen. 2020. No barrier in the road: a comprehensive study and optimization of ARM barriers. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 348–361. https://doi.org/10.1145/3332466.3374535
[32] Weiyu Luo and Brian Demsky. 2021. C11Tester: a race detector for C/C++ atomics. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 630–646. https://doi.org/10.1145/3445814.3446711
[33] Jeremy Manson, William W. Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 378–391. https://doi.org/10.1145/1040305.1040336
[34] Roy Margalit and Ori Lahav. 2021. Verifying observational robustness against a c11-style memory model. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–33. https://doi.org/10.1145/3434285
[35] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 446–455. https://doi.org/10.1145/1250734.1250785
[36] Santosh Nagarakatte, Sebastian Burckhardt, Milo M. K. Martin, and Madanlal Musuvathi. 2012. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 543–554. https://doi.org/10.1145/2254064.2254128
[37] Filip Niksic. 2019. *Combinatorial Constructions for Effective Testing.* Ph. D. Dissertation. Technische Universität Kaiserslautern.
[38] Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 131–150. https://doi.org/10.1145/2509136.2509514
[39] Scott Owens. 2010. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 478–503. https://doi.org/10.1007/978-3-642-14107-2_23
[40] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27
[41] Jean Pichon-Pharabod and Peter Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 622–633. https://doi.org/10.1145/2837614.2837616
[42] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107
[43] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 93–107. https://doi.org/10.1007/978-3-540-31980-1_7
[44] Koushik Sen. 2007. Effective random testing of concurrent programs. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer (Eds.). ACM, 323–332. https://doi.org/10.1145/1321631.1321679
[45] Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2014. Concurrency testing using schedule bounding: an empirical study. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, José E. Moreira and James R. Larus (Eds.). ACM, 15–28. https://doi.org/10.1145/2555243.2555260
[46] Stephen Tu, Wenting Zheng, and Eddie Kohler. 2013. Silo: Multicore in-memorystorage engine. https://github.com/stephentu/silo.

[47] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 18–32. https://doi.org/10.1145/2517349.2522713

[48] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 209–220. https://doi.org/10.1145/2676726.2676995

[49] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. Partial Order Aware Concurrency Sampling. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 317–335. https://doi.org/10.1007/978-3-319-96142-2_20

[50] Xinjing Zhou. 2015. Iris: A low latency asynchronous C++ logging library. https://github.com/zxjcarrot/iris.

Received 2022-07-07; accepted 2022-09-22